SYNCHRONIZATION-POINT DRIVEN RESOURCE MANAGEMENT IN CHIP MULTIPROCESSORS

by

Sokratis Dimitriadis

Diploma in Computer Engineering, University of Patras, 2006

M.Sc. in Computer Science, University of Pittsburgh, 2013

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences,

Department of Computer Science, in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Sokratis Dimitriadis

It was defended on

August 12, 2013

and approved by

Sangyeun Cho, Associate Professor, University of Pittsburgh

Bruce Childers, Professor, University of Pittsburgh

Alex Jones, Associate Professor, University of Pittsburgh (ECE)

Rami Melhem, Professor, University of Pittsburgh

Dissertation Director: Sangyeun Cho, Associate Professor, University of Pittsburgh

SYNCHRONIZATION-POINT DRIVEN RESOURCE MANAGEMENT IN CHIP MULTIPROCESSORS

Sokratis Dimitriadis, PhD

University of Pittsburgh, 2013

With the proliferation of Chip Multiprocessors (CMPs), shared memory multi-threaded programs are expanding fast in every application domain. These programs exhibit execution characteristics that go beyond those observed in single-threaded programs, mainly due to data sharing and synchronization. To ensure that next generation CMPs will perform well on such anticipated workloads, it is vital to understand how these programs and architectures interact, and exploit the unique opportunities presented.

This thesis examines the time-varying execution characteristics of the shared memory workloads in conjunction to the synchronization points that exist in the programs. The main hypothesis is that the type, the position, and the repetitive execution of synchronization constructs can be exploited to unfold important execution phases and enable new optimization opportunities. The research provides a simple application-driven approach for predicting the program behavior and effectively driving dynamic performance optimization and resource management actions in future CMPs.

In the first part of this thesis, I show how synchronization points relate to various program-wide periodic behaviors. Based on the observations, I develop a framework where user-level synchronization primitives are exposed to the hardware and monitored to detect program phases and guide dynamic adaptation. Through workload-driven evaluation, I demonstrate the effectiveness of the framework in improving the performance/power in on-chip interconnects.

The second part of the thesis explores in depth the inter-thread communication behaviors. I show that although synchronization points under the shared memory model do not expose any communication details, they indicate well the points where coherence communication patterns change or repeat. By leveraging this property, I design a synchronization-point-based coherence predictor that uncovers communication patterns with high accuracy, while consuming significantly less hardware resources compared to existing predictors.

In the last part, I investigate the reasons causing threads to wait at synchronization points, wasting resources. I show that these reasons can vary, even across different program phases, and existing critical-thread predictors are ineffective under certain conditions. I then present a new scheme that improves predictability by incorporating history information from previous points. The new design is robust and can dynamically amortize the imbalances to improve the system's performance and/or energy.

Keywords multiprocessors, shared memory, synchronization, dynamic optimization, resource management, varying program behavior.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

To those who know that they know nothing

To those who seek for the truth

Acknowledgments

First and foremost, I would like to thank my advisor Sangyeun Cho for guiding me through the entire journey towards my Ph.D. degree. He taught me how to do research and stay focused, how to give attention to detail, and how to speak and write like a scientist. His kind support and encouragement made this effort truly worthwhile and fruitful.

I would also like to thank Rami Melhem who was like a second advisor to me and always eager to help me. His insightful and well-targeted comments have greatly improved my ability to reason objectively and with clarity. I am also grateful to the rest of my thesis committee, Bruce Childers and Alex Jones. Their comments and suggestions have improved this work tremendously.

Many of the people I've met at the graduate school have supported me in various ways and greatly influenced my research as well as my personality. My fellow members of the CAST lab, Lei Jin, Hyunjin Lee, Kiyeon Lee, Michael Moeng, Ju-Young Jung, and Seungjae Baek, have provided invaluable input and support for my work, and an enjoyable environment for discussion, and collaboration. Mohammad Hammoud, Ahmed Abusamra, and other colleges in our computer architecture reading groups have always been a trustable board for sharing and discussing ideas. I also thank Michel Hanna for the long and fruitful collaboration; Evangelos Vlachos and Michael Papamichael for their support; Onur Mutlu (CMU) for being an inspirational teacher; and Paolo Narvaez (Intel) for being a profound mentor.

I would also like to thank all the friends who have filled my life with nice memories during my

years in Pittsburgh and kept my stay warm and enjoyable. I feel it will be just unfair to simply list names, as it won't suffice to show the unique appreciation and respect I curry for each and every one of you. You are all important to me and you will always be in my thoughts. Special thanks to my roommate Panickos who was always there for me from the very beginning, to my friend Soudi who transformed many dark times to bright ones, and to my dear Samah Mazraani for all her love and care.

Lastly, this dissertation would have never been accomplished without the unconditional love and support of my family. I want to express my sincere gratitude to my parents, Frixo and Anna, who taught me to strive for excellence in all I do and seek a life of virtue. My sincere gratitude also goes to my brother George and my two sisters Eleni and Maria, who have given me tender loving care since the day I was born.

Thank you all.

1.0 INTRODUCTION

The continuing growth of chip density, along with the shortcomings of individual processors (or processor cores) to cope with performance, power, thermal, and complexity limitations, have led chip manufacturers to integrate multiple processors on a single chip instead of simply increasing the complexity of individual cores. Following a decade of advancement in this direction, chip multiprocessors (CMPs) are today a de facto architectural design. Major manufacturers currently ship up to 8-core CMPs for mainstream and high performance systems [50, 49], whereas more specialized chips such as Tilera's TILE-Gx [52] and Intel's MIC [51] already feature 72 and 100 cores. As transistor density continues to grow at an exponential rate in accordance to Moore's law, it is apparent that scaling the number of cores would follow proportionally. Several leading manufacturers have expressed their intention to scale up to hundreds and thousands of cores on a single chip in order to create the backbone of the exascale computing.

The establishment of the CMPs has created a new momentum in the software industry and a driving force in the global economy, pushing the agenda further for commonly practicing parallel computing. First, while parallel computing was primarily a target of high-end systems particular to the scientific domain, it is now widespread in all systems and applications, including the special-purpose domains. Second, the integration of multiprocessors on a single chip has shrunk the communication and synchronization overheads, creating opportunities for scaling and exploiting a more fine-grain thread-level parallelism. Third, the adoption of the shared-memory communication paradigm by the CMPs, along with the tremendous investment in shared-memory software and its programming transparency, has established the shared-memory programming model as the common and dominant programming practice for parallel computing.

While software (programmers and compilers) strives to advance the art of producing good parallel programs for the new era of parallelism, the role of the hardware is to provide the necessary

support for reaching the actual performance limits and ensure that the next-generation CMPs will perform well while effectively managing the valuable on-chip resources. For that, it is essential to understand the important properties of both workloads and architectures as well as how these properties interact. Fundamental goals such as increasing resource utilization, decreasing power consumption, scaling and accelerating communication and synchronization, and orchestrating fine-grain parallelism for load balancing will remain central to this interaction, independent of how the programs will evolve within the shared memory paradigm.

This thesis examines the time-varying execution characteristics of shared-memory parallel applications and the associated optimization opportunities presented, in conjunction to the synchronization points that exist in these applications. The main hypothesis is that *the type, the position, and the repetitive execution of synchronization constructs can be leveraged to produce effective methods for tracking program behavior, and triggering performance optimization and resource management in shared-memory systems*. Our goal is to provide a simple application-driven approach for tracking the varying and repetitive program behavior, and effective in driving dynamic hardware-based resource management strategies in current and future CMPs.

1.1 PROGRAM BEHAVIOR AND SYNCHRONIZATION POINTS

Programs often present vastly different execution characteristics to the architecture over the course of their execution. Program phases with varying resource requirements may lead the system to over- or under-utilization of resources, presumably undermining the potential for power and performance gains. Tracking and detecting repetitive program phases and phase transitions at run time is essential for uncovering opportunities for performance optimization and resource management.

Figure 1 illustrates an example of how bodytrack, a widely used parallel application for tracking human body movements [13], stresses different aspects of the architecture when executed on a 16-core shared memory CMP model. Specifically, it shows the average system throughout (in IPC), the overall L2 cache miss ratio, and on-chip network traffic. The example captures clearly the varying behavior of the application, its variable frequency, as well as its repetitive nature. Once such changes in a program's execution behavior are detected and the stable phases foreseen, ap-

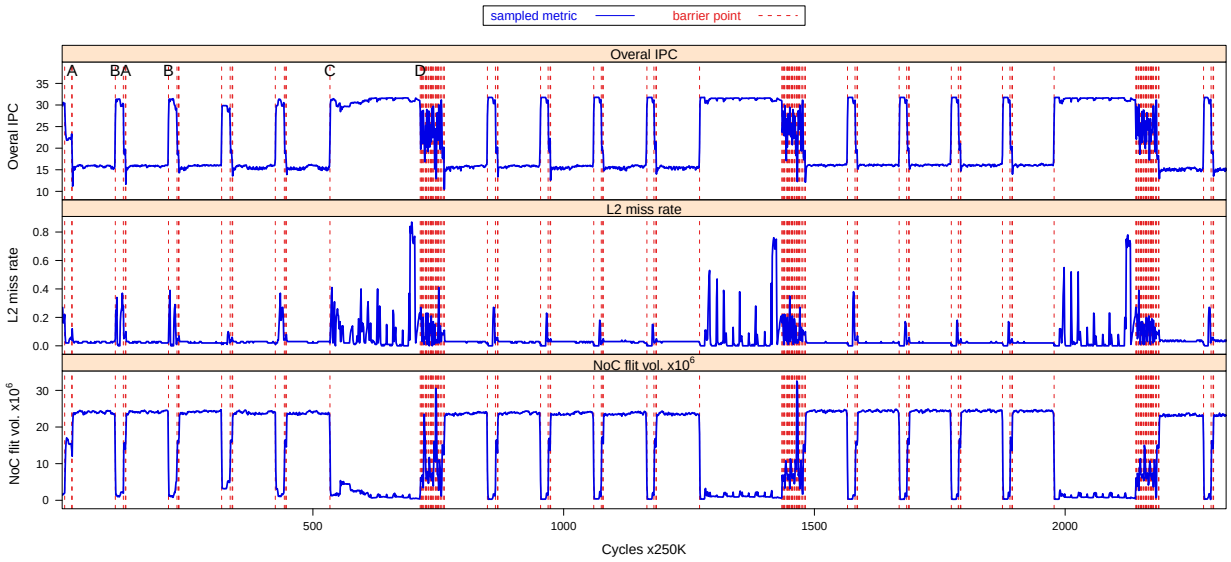

Figure 1: *Key performance metrics of bodytrack:* Measurements were averaged for each sampling interval in time. The sampling interval size is 250K cycles. 'A,' 'B,' 'C' and 'D' are examples of repeating global synchronization points in the program.

propriate decisions can be made to adapt the system to a performance and/or power optimal state. For example, decreasing the available bandwidth or the operational frequency (and voltage) of the on-chip network during low volume periods could possibly save considerable power without compromising performance.

A system can sense shifts in program behavioral either by continuously monitoring certain metrics to catch a "significant" change (e.g., IPC, miss rate, number of branches), or by using some hint from the program itself indicating that a change is about to happen. Typically the former approach, although transparent, works in a reactive manner and lacks a higher-level understanding of the program behavior and its periodic nature [87, 58, 101, 113, 5, 39, 63, 8, 30, 106, 57]. Software hints offer instead a more proactive and structural approach to uncovering the program phases; however, they usually require application profiling and considerable effort by programmers and compilers to define and embed software constructs in the program binary [105, 85, 84, 104, 72, 96]. Also, as the progress of co-executing threads is often non-deterministic, software hints coming from different threads might be often rendered ineffective.

A distinctive and fundamental aspect of all parallel applications is synchronization—the need

to ensure that the relative progress of concurrently-executing threads is consistent according to a memory consistency model. In the shared memory model, synchronization is explicitly enforced through software primitives to ensure either mutual exclusion, or event-based synchronization. Event synchronization (e.g., global barriers and point-to-point conditions) is often used in a structural way, naturally separating distinct sections of computation in the program [24]. Therefore, it is possible to be closely related with the underlying changes in program behavior. The vertical dotted lines in Figure 1 represent the global synchronization points (barriers) that exist in the application. It is clear from the figure that those points are well aligned to transitions in program behavior and they define intervals that can capture the anticipated length and repetitive nature of different program phases.

Motivated by this fact, I perform a more in depth investigation on how synchronization points can be used as hints to systematically detect and predict program behavioral transitions and patterns at run time. Through statistical analysis, I show that application-wide time-varying behaviors can be effectively tracked and predicted in globally-synchronized intervals. For example, alternating phases of traffic volume such as those illustrated in Figure 1 are well captured and their periodic behavior well understood. By dynamically adapting the operating voltage and frequency of the on chip network routers, the system can exploit the network volume variations and improve its power/performance trade-off. In addition, I explore optimization opportunities in behaviors that are naturally related to synchronization points such as coherence activity patterns between cores, and idle times due to behavioral imbalances across cores. These optimizations are discussed separately throughout the chapters of this thesis.

The concept of tracking synchronization points to create a certain understanding of the program's execution characteristics is not new. Particularly in the message-passing programming interface (MPI), examining the synchronization points (MPI calls) is common as they explicitly separate computation tasks from communication activity [110, 100, 99, 36, 82, 38]. In the shared memory model, however, no such distinction is visible and therefore communication patterns are less obvious in such points. Also, random delays and unpredictable slack times become more prevalent across co-executed threads. Related work in the shared-memory domain has specifically attempted to improve the idle cycle times occurring due to synchronization [80, 83, 112].

1.2 COHERENCE COMMUNICATION PREDICTION

The frequency, speed, and the way in which processors communicate in parallel execution are among the fundamental attributes in achieving optimal parallel scaling. Shared-memory systems offer a simplified communication abstraction to the programmer, while delegating to the hardware the responsibility of implementing the communication layer. Communication is therefore automatically generated for cache coherency and occurs when data requests must contact at least one other processor in order to be correctly satisfied. In directory-based protocols, requests must be directed to the directory to first discover their destination, adding considerable extra latency for the request to be completed. Snooping protocols avoid this indirection by broadcasting the requests to all processors; By doing so, however, they place significant bandwidth demands on the interconnect, even for a moderate number of processors [88].

A common approach to improving coherence communication is to predict the processors to which a coherence request must be delivered to, avoiding therefore an indirection or broadcasting. Such predictions can be made by programmers (e.g., [41, 1]), compilers [67, 116], or transparently by the hardware [75, 92, 69, 61, 15, 70, 2, 3, 88, 20]. Given that compiler techniques are limited to static optimization [67] and that the shared-memory model should be kept transparent while offering high performance [92], hardware-based approaches with fine-grain dynamic prediction capabilities have been mostly preferred despite the relatively large cost in space requirements.

In this thesis, I propose a *synchronization point based prediction* of coherence communication, a run-time technique that exploits the location and type of synchronization points to predict coherence request targets. The proposed predictor builds on the intuition that inter-thread communication caused by coherence transactions is directly linked to the synchronization points in parallel execution. By dynamically tracking communication behavior across synchronization points and uncovering important communication patterns, the new prediction approach associates those patterns with each synchronization point in the instruction stream to predict the communication of requests that follow each synchronization point.

The intuitive relation between synchronization points and the communication patterns comes from the fact that those points actually exist to ensure the correctness of the shared-memory communication abstraction. As a result, although the software synchronization mechanisms tell noth-

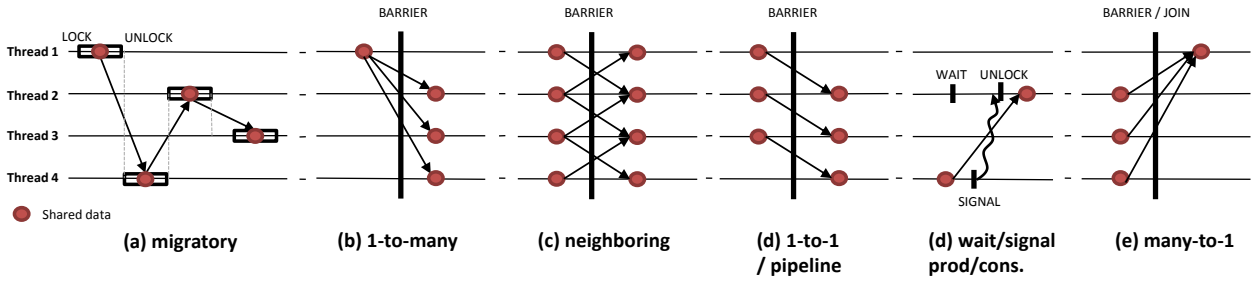

Figure 2: *Various examples of data sharing as observed across synchronization points:* The arrows illustrate the communication effects of data sharing.

ing specific about the underlying communication, they naturally indicate the points at which certain data private to a processor become visible and possibly communicated to other processors. Figure 2 shows a number of different examples of how data are being produced and consumed by different threads. Independent of how the data sharing and communication is realized, synchronization points appear to be a common characteristic naturally related to the communication activity.

The proposed prediction scheme is different than existing shared-memory hardware prediction techniques because it exploits the synchronization constructs that are inherent to the application itself to predict communication patterns. The new approach creates a new scope of application-defined temporal predictability which has low implementation cost and hardware resource usage and yet delivers relatively high performance.

1.3 THREAD CRITICALITY PREDICTION

Synchronization points inherently cause threads to wait for each other, stalling program progress, limiting performance, and wasting energy. For example, at the barrier points, threads typically wait for the slowest (a.k.a. *most critical*) thread yet to reach the barrier. In the case of a critical section, a thread might cause other threads to wait by holding the lock that other threads try to acquire as well. Identifying critical threads or threads with long serial sections is important as those threads largely determine the overall performance of the application and thus can be targeted

for performance optimization [80, 83, 112, 112, 29, 12, 17, 71].

Unequal completion times across threads occurs for several reasons: Unequal work assigned to different threads, unequal data localities, heterogeneous hardware and functional parallelism are some of the obvious ones. Further to that, thread progress is affected by other random architectural effects that occur during execution such as delays due to contention on shared resources, communication, lock serialization, and interference from system software. Although the goal of software load balancing is to reduce the time threads spend waiting at synchronization points, the random effects, along with the several trade-offs related to the task granularity and scheduling policies make perfect balancing impossible. Several hardware and software dynamic techniques have been proposed to further amortize the execution imbalance across threads and optimize in a finer granularity. Core DVFS, thread migration, and thread-fetch priority are some of the mechanisms that can enable such techniques.

Identifying or predicting critical threads before they emerge is essential for dynamically balancing the relative progress of threads. Prior research has shown that critical threads can be effectively detected and predicted by tracking simple metrics such as relative cache miss rates [12], instructions executed [71], "useful" progress [29], and other related metrics [80, 83, 17]. Assuming threads execute identical tasks on identical processors, the use of these metrics could reasonably provide accurate predictions. However, as threads compete for limited shared resources, and as parallel programming practices and architectures encourage more and more heterogeneity in both computation and resources, parallel processing becomes more diverse and expected behaviors less common.

In this thesis, I investigate thread criticality and characterize its possible sources in various barrier-based workloads. With quantitative examples I show that the impact from the different sources can significantly vary across different workloads and architectures, as well as across the execution phases of the same program. Thus, predicting critical threads by relying only on a particular metric that captures only part of the possible reasons can lead to results that are not robust across different workloads and architectures.

By using information kept at the barriers, I show that it is possible to improve the phase-level knowledge about the relative load across threads and build criticaliy predictors which are adaptive, robust and highly accurate across the workload phases. The proposed mechanism predicts thread

criticality by estimating the actual time each thread needs to hit the next barrier without relying on the assumption that threads have equal work, but on the fact that threads repeat the same relative work. This provides a significant advantage over previous temporal adaptive predictors as it tracks criticality on its natural granularity, accounts for the overall random delays using a simple metric, and is able to dynamically tolerate any inherently unbalanced loads or heterogeneous hardware.

1.4 CONTRIBUTIONS

This thesis presents a new framework for driving dynamic optimization and resource management techniques in future chip multiprocessors. The new framework leverages shared-memory synchronization semantics to improve the knowledge about program's execution behavior and produce performance-, power-, and cost-effective dynamic resource management decisions. In summary, the principal contributions are the following:

- *Characterization of program behavior across and within synchronization-point defined intervals.* I examine application-wide time-varying behaviors of shared memory programs in conjunction to synchronization-defined execution intervals and I quantify the inherent relations that exist between them. The analysis highlights the fact that these intervals form simple phases of the program execution, and identifies certain related optimization opportunities.
- *Synchronization-point phase-based dynamic adaptation of the on-chip network.* Through cycle-accurate simulation and power consumption modeling I show that by dynamically adjusting the voltage and frequency of the on-chip network (NoC) in barrier-defined intervals it is possible to considerably improve the overall performance-energy trade-off of the system. To the best of my knowledge, this is the first study that performs such a simple, yet effective, phase-based adaptation of an on-chip shared resource at this granularity. The applied methodology forms a framework for synchronization-point driven dynamic adaptation in CMPs.
- *Synchronization-point-based coherence prediction.* I design a new technique to accurately predict the destination of each coherence request in multicore/multiprocessor system by leveraging dynamic information kept at synchronization points. The new design uses less hardware

resources compare to existing techniques, yet achieves comparable or better accuracy than existing techniques. The design is shown to accelerate communication between cores and in turn reduce the average miss latency in a directory-based coherent CMP.

- *Robust thread criticality prediction.* I show that by exploiting the repeatability of barrier-defined intervals, the execution imbalances between threads can be more accurately identified and predicted. Based on this observation, I propose a new thread criticality predictor that leverages this information to predict the most critical thread with high confidence, before any of the threads reaches the upcoming barrier. The corresponding critical core can then be accelerated to reduce the performance cost of criticality, or the rest of the cores slowed down to reduce the wasted cycles of waiting the critical core.

1.5 THESIS ORGANIZATION

The rest of the thesis is organized as follows. Chapter 2 discuss in general the synchronization points and their associated execution intervals and gives the necessary preliminaries. Then presents observations related to global synchronization points and how those are related to the program behavior. The conclusions are highlighted through applying synchronization-point driven adaptation of the NoC. Chapter 3 extends the ideas described and introduces and evaluates a scheme that can predict coherence communication with unique advantages over existing approaches. Chapters 4 examines in depth the problem of thread criticality and presents and evaluates a new criticality predictor. Finally, concluding remarks are outlined in Chapter 5.

Most of the material included in Chapters 2 and 3 was previously presented in CF 2011 [26] and MICRO 2012 [27]. Material presented in Chapter 4 has not yet appeared in other venues.

2.0 TRACKING PROGRAM BEHAVIOR AND SYNCHRONIZATION-POINT DRIVEN DYNAMIC ADAPTATION

2.1 INTRODUCTION

Many programs exhibit widely different behavior over the course of their execution [105]. For example, during one part of the execution a program can be compute intensive; in another, it can be memory intensive, increasing the bandwidth pressure on the memory bus. Average statistics gathered for a program might not accurately reflect its real behavioral characteristics. This realization can lead to many new opportunities for dynamic performance optimization and resource management, both in hardware and software. As a basic requirement to exploiting the time-varying characteristics is the ability of a system to track and discover "phases" in program behavior and adapt to them appropriately.

There have been several techniques proposed to allow the rapid low-cost adaptation of hardware resources to match the application's varying performance and power requirements [8, 30, 106, 57, 53]. Many of them are empowered through dynamically adapting the operating voltage and frequency of processors [87, 58, 101], interconnects [64, 102, 90] and other hardware components(e.g., [25]); in other cases through reconfiguring certain hardware structures (e.g., adaptive caches [113], morphable processors [5, 39, 63], etc.); or even by migrating load to different cores to improve performance (e.g., code migration in heterogeneous CMPs [112]). There have been also several feedback-directed software-based adaptive approaches. Some of them can make use of the same management techniques under a coarser granularity control, while other aim run-time resource allocation and task scheduling policies [85, 46, 37, 97].

Most adaptation techniques operate in a reactive manner [87, 58, 101, 113, 5, 39, 63]. Decisions to adapt a system to a new state are taken after some change in the program behavior has

been observed. The immediate past behavior is usually used as representative of future behavior in order to perform the appropriate adaptation action. Reactive approaches work well as long as programs execute in a sequence of more or less stable phases with relatively few transitions. However, if a program exhibits significant oscillations in its behavior, a reactive approach may mislead the system in adapting for behaviors that have already passed. To address this issue, some proposals suggest the use of history-based predictors that allow more confidence in recognizing an upcoming stable phase or phase transition. These approaches remarkably exploit the repeatability of program behavior and can classify an observed behavior as part of a phase that has been previously seen and recorded. Nevertheless, all sample-based approaches heavily depend on the length of the sampling interval and the sensitivity of the classification algorithm which commonly rely on empirical observations.

Alternatively, a number of techniques use offline profiling to extract the phase representation of a program and use it to hint the phases during future execution of the program [105, 85, 84, 104, 72, 96]. Although these techniques address the problem in a proactive manner, on the most part they require extensive offline profile effort and tools, are often tight to the problem size being profiled (and in sometime on the machine), and require special markers to be injected in the code along with binary rewriting and special support by the machine. For these reasons, these techniques have been mostly popular in simulation practise for creating simulation points of interest.

Shared-memory multithreaded workloads exhibit dynamic behavior which is further affected by the sharing of data and platform resources [107]. As a result, traditional phase detection techniques may often mislead to wrong conclusions due to the effects caused by frequent random oscillations caused by inter-thread interference and resource contention. In addition, if the metrics being monitored are based on the code itself, the relative progresses of co-scheduled threads may result in an inconsistent global program phase representation and thereby hamper phase detection. Prior work has identified these challenges; however, the few attempts addressing the problem are limited to profile-based phase detection algorithms.

In this thesis, we introduce a new approach for driving dynamic adaptation, optimization and resource management in shared-memory systems that uses the synchronization primitives in parallel application as hints for phase transition and prediction. The rationale behind this idea is that synchronization points often define sections of code that perform a unified, logically distinct tasks

that are being replayed throughout the execution, and naturally indicate a shift from a certain program phase to another. We show that although this approach is simple in concept, is adequately effective in guiding dynamic adaptation.

The direct use of synchronization points as hints to discovering phase transitions at run-time is particularly attractive as it combines the advantages of an application-driven proactive approach with simplicity and ease of implementation. Specifically, synchronization points are (1) code-related and thus can be identified independently of the architecture; (2) they are able to instantly recognize a potential phase transition and predict the stability of the next phase without the need to sample and classify intervals; (3) they indicate the only points in program execution that can deterministically capture global and individual thread state, and allow conclusions about the relative behavior of threads; (4) they come for "free" as they inherently exist in many parallel applications; and (5) they can be easily exposed to the hardware and the run-time system, and easily identified.

The concept of tracking program execution at a granularity defined by synchronization points is not new. Work in the high performance community that studies workloads under the message-passing programming interface has often used MPI calls to model the program's task flow [110, 100, 99, 36, 82, 38]. Note however that in the case of message passing programs, MPI calls explicitly separate computation from communication tasks;[1] therefore this distinction has mainly serve for the analysis of the compute and slack times spent in computing phases, and the communication patterns that appear during communication phases. In shared memory, no such distinction is visible and therefore the same analysis is inherently more difficult and the predictable components not obvious. Chapters 3 and 4 discuss in depth the particularities in analyzing and predicting communication patterns and relative slacking times as they appear in the shared memory model.

In this chapter, we focus on (1) characterizing various aspects of the program behavior within and across the synchronization-defined intervals, and (2) demonstrating how synchronization points can effectively drive DVFS on the chip interconnect to improve the overall energy of the system without compromising performance. Our results suggest that synchronization-point driven resource management is a simple approach that can effectively capture various important and repet-

[1]Note that "task" is a broad term used to define a certain piece of code and its dependencies, and is not restricted to synchronized intervals–which can essentially include many tasks. In shared-memory programming interfaces, tasks can be created at run-time. Also note that critical sections (assumed to also be synchronization-defined intervals) do not necessarily define distinct tasks.

itive phases in program execution and consequently find applicability in various dynamic performance and power optimization mechanisms.

2.2 BACKGROUND AND RELATED WORK

2.2.1 Tracking Program Behavior and Dynamic Optimization

The most common approach for tracking stable phases or phase transitions during program execution is to dynamically track a certain metric on fixed-sized continues intervals, and sense whether the behavior in a given interval appears to be similar or different compared to the previous interval. A simple reactive algorithm can then assume that the future behavior will follow the immediate past behavior and use this to adapt the system on the fly. The choice of what metric to monitor depends on the targeted optimization, where the sensitivity of the measurements towards the metric and the length of the sampling interval usually depend on analyzing the related trade offs and on empirical observations.

Balasubramonian et al. [8] use hardware counters to measure miss rate and branch frequencies and they identify a behavioral change by comparing them with thresholds that are adjusted dynamically. Their method is used to guide dynamic cache reconfiguration to save energy. Dhodopkan and Smith [28] use "working set signatures" to characterize each interval of execution and identify a phase change by calculating a signature distance between consecutive time intervals. They assume that the program enters into a stable phase only after identical behavior is observed across a number of consecutive samples. To reduce re-optimization overhead whenever a new phase is detected, they store a configuration for every signature and reinstate it when intervals with the same signature occur. This essentially exploits the repeatability of program behavior, which is prevalent during program execution. Based on the observation that the program behavior repeats according to the code executed, Sherwood et al. [106] present a dynamic phase tracking approach that classifies each interval based on the basic blocks that are touched during the interval. Thus, intervals exercising roughly the same code are expected to have the same behavior and are classified as similar. In addition, they present a history-based predictor that allows a more powerful

detection of stable phases or phase transitions based on the history of already observed phases. Lau et al. [73] investigate other code-related metrics for capturing program phases, such as loop-, procedure-, or register-usage vectors. Duesterwald et al. [30] introduce a "cross-metric" predictor that uses one metric to predict another. Using software-based adaptation, they highlight the fact these predictors are powerful in detecting and predicting program phases using performance counters. Isci and Martonosi [57] and Peleg and Mendelson [94] have further shown the potential of using hardware performance counters to dynamically monitor and optimize for power-aware and SMT architectures respectively. Sampling intervals have been studied in various scales, e.g., from 100K [8] to 10M instructions [105]. Vandeputte and Eeckhout [117] present "phase complexity surfaces" as a method to systematically characterize a program's phase behavior in different time scales.

An alternative approach for detecting program phases at run-time is to track regions of code that have been pre-determined as distinct phases and can expose this information at run-time. Typically, such approaches rely on static profile-based methods to determine the regions of interest, and inject markers in the binary that can hint certain information at run-time. Sherwood et al. [105] create an interval-based representation of program behavior based on basic block vectors (BBVs) and then employ off-line clustering to classify regions of program execution into phases. This approach has led to a popular and successful approach for creating simulation points. Magklis et al. [85] use static profiling to discover long-running subroutines and loops, analyze them and select the most appropriate for reconfiguration. The configurations settings are then made available at run-time through binary rewriting. Liu et al. [84] use a similar profile-based approach to select representative subroutines and loops to speed up detailed simulation. Many other studies also use profiling to identify various locations in the code that indicate phase changes, and propose to incorporate this information into the program binary as software phase markers [104, 72, 96].

Huang et al. [53]–and previous to that Balasubramonian et al. [8]–have proposed a region-based adaptive approach that is well tailored for dynamic optimization. Specifically, they use subroutines as a region and they show that the repetitive program behavior can be effectively tracked and exploited at this granularity. They use a hardware call stack to identify major program subroutines and look for program changes by comparing the program behavior across different subroutines. By keeping the best configuration for each subroutine, they are able to effectively adapt the

system in later iterations of the same subroutines for energy savings.

In comparison, the method presented in this work is closely related to the subroutine-based approach by Huang et al. [53] as is also proposing a region-based approach that is based on inherent code markers and requires no offline profiling or binary rewriting. However, our approach instead uses as region boundaries synchronization-points. These points unfold a different structure and granularity and are likely to be more well suited for the shared-memory application domain. For example, in multithreaded environments, different threads may execute code from different subroutines at any point in time, and therefore may cause an inconsistent global program phase representation across their repetitive instances. In fact, many of the targeted optimizations presented in this work, such as communication and thread criticality, build rather more naturally on synchronization-based intervals than subroutines. Lastly, the two approaches can be combined to provide orthogonal improvements, especially in cases where long subroutines or synchronization intervals include sub-phases that remain exploited.

All the aforementioned work specifically targets single-threaded programs. Applying these approaches for multithreaded programs is theoretically possible, although there are new challenges involved. Some recent efforts have pointed out some of the inaccuracies that result when using these approaches in multithreaded programs. Proposed solutions extend existing techniques for profile-based phase detection algorithms [95, 120].

2.2.2 Prior Work using Synchronization Points for Run Time Optimizations

The concept of using synchronization points to direct optimizations has been previously used in the message-passing programming model (MPI) [110, 100, 99, 36, 82, 38], as well as in some other more recent works targeting shared-memory applications [80, 83, 112, 112]. In the case of MPI, synchronization explicitly separates computing tasks from communication actions; thus tracking the execution time between and within MPI calls is often used to model the program's a task flow graph where MPI calls determine graph dependencies and their weights. Several works have used this model to estimate (offline or online) slack times across threads and amortize them through DVFS techniques. In Chapter 4 we describe and address the same problem but in the shared-memory application domain, where we also discuss further related work.

A more recent work by Ioannou et al. [56] uses MPI calls to guide phase-based power management for Intels MPI Cloud processor research prototype [44]. The ideas are similar in concept to the our synchronization-based power management approach as presented in this chapter. However, our work differs by targeting the shared memory model, where phases are less evident compared to the MPI model. Also note that our work precedes this related work.

In the shared memory domain, exposing synchronization primitives to the hardware has been an inherent requirement in machines implementing release (and other more relaxed) memory consistency models. These have served as the underlying implementations for supporting software coherence [19, 22]. In other more recent works, locks and other synchronization points are tracked by the hardware to trigger code migration to a faster core in order to accelerate critical sections [112] and other critical bottlenecks [60]. Lastly, work on memory scheduling for parallel applications have illustrate the use of loop-based synchronization to effectively manage inter-thread DRAM interference [31].

2.3 SYNCHRONIZATION EPOCHS

2.3.1 Synchronization Points

Parallel, shared-memory programs require synchronization to coordinate race conditions between concurrent threads and ensure that operations on shared memory locations are consistent according to a specified memory consistency model.

Synchronization is enforced by inserting software synchronization primitives into the code, as indicated (explicitly or implicitly) by the programmer. The primitives are typically implemented as library calls or macros. Synchronization libraries (e.g., POSIXthreads and OpenMP) offer various synchronization operations to modern parallel programming environments, although they might differ on API semantics and provide different level of abstraction. Throughout this work, we assume POSIX thread primitives, which are typical of low level programming environments; however, the concepts presented are applicable to other implementations as well.

A synchronization point (*sync-point*) is any point in execution at which a software synchro-

BENCHMARK	SUITE .	DESCRITPION .	PROGRAM INPUT	BARRIERS (STATIC)	LOCKS (STATIC)	COND. (STATIC)	BARRIERS (DYNAMIC)	LOCKS (DYNAMIC)	COND. (DYNAMIC)
barnes	Splash-2	N-body Simulation (3-D)	64K (particles)	3	2	-	9	8,598	-
fmm	Splash-2	N-body Simulation (2-D)	64K (particles)	10	10	-	34	11,588	-
lu	Splash-2	Matrix Triangulation	2,048 (matrix)	5	7	-	259	260	-
ocean	Splash-2	Ocean Current Simulation	1,026 (grid)	20	28	-	708	870	-
radiosity	Splash-2	Graphics Iterative Diffusion	largeroom	3	34	-	18	137,172	-
water-ns	Splash-2	Molecular Dynamics	1,000 (mol.)	9	20	-	20	6,207	-
cholesky	Splash-2	Matrix Factorization	tk15.O	-	28	4	-	2,555	115
fft	Splash-2	Signal Processing (1-D fft)	256K (points)	7	8	-	7	8	-
radix	Splash-2	Integer Sort	4M (keys)	7	8	7	11	12	31
water-sp	Splash-2	Molecular Dynamics	512 (mol.)	9	17	-	20	39	-
bodytrack	Parsec	Computer Vision	simlarge	3	16	4	20	251	102
fluidanimate	Parsec	Animation	simsmall	8	11	-	40	223,641	-
streamcluster	Parsec	Data Mining	simlarge	21	1	2	11,397	38	22
vips	Parsec	Media Processing	simlarge	-	14	7	-	1,384	167
facesim	Parsec	Animation	simlarge	-	2	-	-	1,976	-
ferret	Parsec	Similarity Search	simlarge	-	4	-	-	47	-
dedup	Parsec	Enterprise Storage	simlarge	-	3	-	-	6,222	-
x264	Parsec	Media Processing	simlarge	-	2	-	-	1,875	-

Table 1: Synchronization-point statistics of benchmarks. The numbers were collected on a 16-core and correspond to per-thread median statistics.

nization routine is invoked. There are 3 basic synchronization operations: Mutual exclusion (e.g., locks), global synchronization (e.g., barriers), and point-to-point synchronization (e.g., condition variables). Each sync-point has a type, e.g, `barrier`, `join`, `wait`, `broadcast`, `signal`, `lock`, and `unlock`, and a static and dynamic ID. The static ID identifies each sync-point statically in the program code and corresponds to its calling location (e.g., program counter). The dynamic ID of a sync-point can be expressed with the corresponding static sync-point ID and how many times it has been executed so far. Hence, the dynamic ID uniquely identifies the multiple dynamic appearances of sync-points that have the same static ID.

Table 1 lists basic synchronization point statistics for the commercial and scientific multi-threaded programs that are examined in this work. The programs are taken from the PARSEC 2.0 [13] and SPLASH-2 [119] benchmark suites. These programs use *pthreads*, which is a standard shared memory programming library that provides interface to all the basic synchronization operations. The statistics for each program include the number of distinct sync-points appearing during execution as well as the number of their dynamic appearances.

2.3.2 Synchronization Epochs

A *Synchronization Epoch (sync-epoch)* is an execution interval enclosed by two consecutive synchronization points. According to this simple definition, on each sync-point a new sync-epoch starts and the previous sync-epoch ends. A sync-epoch is identified by the type and the static IDs of the two enclosing sync-points, and a dynamic ID that shows how many times has been executed so far. A static sync-epoch that is exercised multiple times during execution creates *dynamic instances* of itself. Consequently, the execution of each thread can be viewed as a sequence of dynamic instances of sync-epochs. More formally, if E_x denotes a sync-epoch with static ID x and R the number of unique sync-epochs in a thread, then the execution of the thread can be decomposed to the following dynamic sets of epochs:

$$E_1 = E_{1,1}, E_{1,2}, \ldots, E_{1,N_1}$$
$$E_2 = E_{2,1}, E_{2,2}, \ldots, E_{2,N_2}$$
$$\ldots$$
$$E_R = E_{R,1}, E_{R,2}, \ldots, E_{R,N_R}$$

Each sync-epoch E_x has N_x instances, which appear in-order with respect to the other instances that belong to the same epoch. Note that instances that belong to different epochs can appear in any order.

The exact definition of the sync-epoch can be adjusted by restricting which type of sync-points will participate in forming them. In general, sync-epochs formed between any type of sync-points are said to define *thread-level epochs* as each threads may exhibit a different number and type of sync-points. Alternatively, sync-epochs formed strictly between global synchronization points (i.e., barriers-based epochs) define *application-wide, or global-epochs*. A critical section is simply a sync-epoch that begins with a lock and ends with an unlock and appears under thread-level epochs. Figure 3 depicts different global sync-epochs and the notion of a static and dynamic ID.

Global and thread-level epochs are both examined throughout the thesis. Global-epochs are particularly interesting when examining system-wide program phases, or when examining the threads' execution progress towards global synchronization points. Thread-level epochs, which

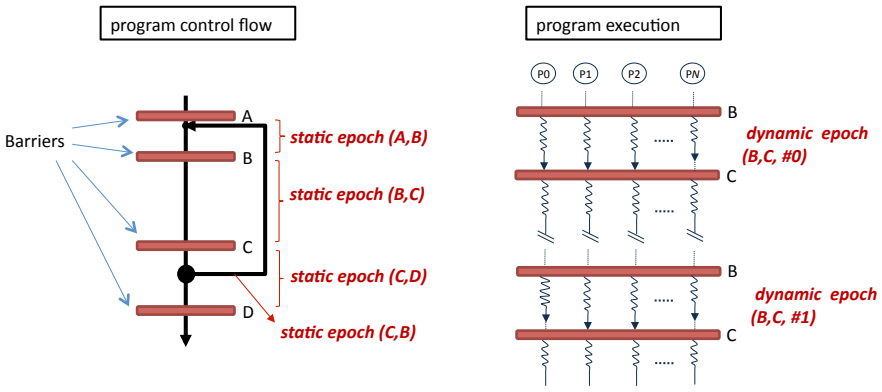

Figure 3: Static identification of epochs (left) and dynamic epoch instances (right).

instead define finer intervals, are more interesting when examining thread/core-level behaviors or core inter-dependent behaviors (e.g., coherence communication).

Sync-epoch statistics are directly linked to the sync-point statistics, which are listed in Table 1. Specifically, the number of dynamic epoch instances is expected to be equal to the number of dynamic appearances of sync-points. Although the same often holds for the number of static sync-epochs, there are cases where sync-epochs are more than sync-points. This happens due to multiple execution paths that often exist between sync-points. More sync-epoch statistics and discussion can be found later in this section and throughout the thesis.

2.3.3 Global Epochs in Multithreaded Programs

This chapter focuses on the global synchronization points (`barrier`) and examines specifically the global sync-epochs i.e., the intervals enclosed by two consecutive barriers. For simplicity, we will refer to global sync-epochs as simply *epochs* throughout this chapter.

Barriers are usually a conservative way of preserving dependencies; however, from a programming point of view, they are more convenient to use and therefore are quite common in parallel programs. Furthermore, barriers are usually used in a structured way, naturally separating distinct phases of computation in the program. Therefore, the intervals enclosed by barrier sync-points are particularly interesting when the program time-varying behavior is under examination.

BENCHMARK	# STATIC EPOCHS	REPEATED EPOCHS	PROGRAM INPUT	# DYN. EPOCHS
bodytrack	4	4/4	simlarge	19
fluidanimate	8	8/8	simlarge	39
streamcluster	21	16/21	simmedium	7,569
barnes	3	2/3	64K (particles)	8
fmm	10	6/10	64K (particles)	33
lu	5	2/5	2,048 (matrix)	258
ocean	24	18/24	1,026 (grid)	707
radiosity	5	2/5	largeroom	17
water-ns	11	5/11	1,000 (mol.)	19

Table 2: Global-epoch statistics of benchmarks.

A good example that shows how barriers are used in a structured way is bodytrack, which is a computer vision application designed for tracking human body movements [13]. This example was also mentioned in Section 1.1 and Figure 1. In this application, there are three parallel kernels which are separated by barrier points: (1) the edge detection, to detect the edges of the body; (2) the edge smoothing, a Gaussian filter for smoothing the edges; and (3) the part that calculates/computes a weight for each body's particle. The barriers between these kernels are necessary to ensure correct synchronization and load balance between the parallel threads. Furthermore, the filtering kernel, which consists of two distinct parallel phases, includes additional barriers to synchronize the matrix traversals across these phases.

Since each kernel performs a logically distinct task with possibly distinct resource requirements, barriers are possibly linked with the changes in program behavior, partitioning the program into its main phases. If this is true, barrier primitives could then be used as effective indicators for run-time performance optimization and resource management. This information is inherent in the program code itself; however, it is not visible to the hardware at run time.

Table 2 lists basic global-epoch statistics for each program. Besides the number of static and dynamic global epochs of each program, the table also shows the fraction of those epochs that appear more than once. Note that Table 2 does not list programs that do not have epochs (e.g., do not use barrier synchronization, or have barriers that do not reside inside repeatable sections).

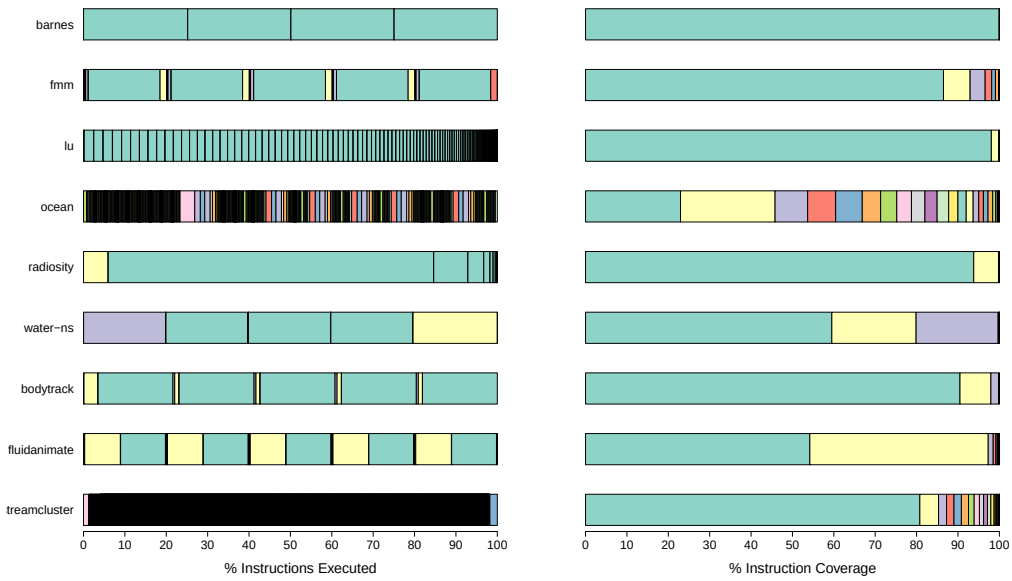

Figure 4: *Relative epoch size (as a percentage to the total instructions executed)*: Bars on the left show all epochs and their instances in execution order. Bars on the right show the total amount of instructions consumed by each unique epoch, with all its instances merged together. The different colors correspond to different unique epochs. The dark areas are due to very dense epoch transitions.

`Fork` and `join` points (if they exist) also participate in forming epochs since are viewed as barrier points; however, a fork point can only indicate the beginning of an epoch, whereas a join point only the end of an epoch. Our characterization in this chapter applies only to shared-memory parallel programs that use barriers.

The table shows that the most of epochs repeat. Some programs have a large number of dynamic epochs generated from a few epochs, such as *ocean*, *lu*, and *streamcluster*, because these epochs reside inside heavily repetitive loops. Other programs have epochs with few dynamic instances, each with probably a long execution times. Epochs that do not repeat usually enclose very long sections of code that spans across many functions and loops, or very small sections at the beginning or the end of parallel sections, or in between other epochs.

We extend the profile of program's epochs with statistics related to the amount of instruction executed by each epoch in order to give a sense of their relative "size" and "importance". Figure 4 shows the relative number of instructions executed by each epoch instance in execution order,

which gives a sense of the relative "size" between epochs. On the right side, the same figure shows the relative number of instructions executed in total by each unique epoch, indicting the "importance" of each epoch. An epoch with a large size does not necessarily reflect an execution-dominant epoch, since its total contribution to the overall execution depends not only on its size, but also on the number of times it is executed.

The profile results suggest that some programs such as *ocean* and *streamcluster* spend most of their time in a number of small epochs that are highly repetitive. Other programs (*fmm*, *fluidanimate*, *water-ns* and *bodytrack*) spend a large fraction of their time in a single epoch, while the remaining time is split rather equally to the rest of the epochs. Finally, in *barnes*, *lu* and *radiosity*, a single static epoch monopolizes the execution time. A main difference between barnes lu and radiosity, however, is that the dominant epoch varies in the number of iterations. For example, barnes and lu have many iterations, whereas in radiosity the dominant epoch runs just once. The large range of epoch granularity reflects the structural heterogeneity across and within different programs. At the two extremes, very fine or very coarse epochs may exhibit unstable or multi-phase behavior, respectively. The epoch-level characterization in the following sections presents in detail the program behavior across and within epochs, and shows the promising aspects of our approach.

2.4 EPOCH LEVEL CHARACTERIZATION

2.4.1 About Repetitive Program Behavior

Program behavior is strongly related to the code that is executed. Given that epochs repeat throughout the program execution and that their instances are likely to exercise the same or similar code, it is highly possible that similar behavior and predictable patterns would occur at the epoch granularity.

We track the repetitive nature of epochs in the program and we characterize the behavioral similarity across the dynamic instances of each epoch while indicating the differences across completely different epochs. The primary objective is to derive conclusions on whether epochs provide

a predictable representation of program phases, and whether this representation is good enough to effectively guide phase-based optimization and resource management at random.

We note here that the conceptualization of epochs as program phases does not follow and should not be confused with the conventional concept of a program phase, where phases and phase transitions should obey well-defined granularity and similarity parameters [42]. Epochs naturally partition the program code into unique, contiguous, variable-length intervals. Marking epoch boundaries is done automatically, without relying on similarity parameters, and without a need for run-time monitoring, profile- or compiler-based analysis. Consequently, epochs do not aim at guaranteeing behavior stability and transitioning points, but rather providing a simple, lightweight yet effective approach for guiding phase-based optimization and resource management at run-time.

2.4.2 Methodology

Performance Metrics and Variation Analysis.

We characterize the program behavior by tracking system-wide metrics. Specifically, we measure system throughput (IPC.total), L2 miss ratio (L2.Miss), on-chip interconnect traffic volume (Traffic.Vol), and cache-to-cache transfer hit ratio (C2C.Hit). NoC traffic includes memory controller requests, remote L2 requests, cache-to-cache requests and all control and coherence messages. C2C-transfer hit ratio is defined as the fraction of L2 misses that were satisfied on chip, by the cache-to-cache transfer protocol.

For each epoch E_x, we denote the mean and variation of a metric across the dynamic instances of the epoch. Since the metrics can be considered as interval metrics,[2] we use standard deviation and we measure the variation as a percentage. Different variations around different means can be directly compared. We exclude from the analysis all epoch instances that are too short (we consider 50K dynamic instructions to be the threshold). This is because very short time intervals cannot reach a stable and unbiased microarchitectural state (i.e., they only experience the warm-up period); therefore, the performance metrics cannot precisely capture their actual characteristics and may introduce "noise" to the analysis. Exceptions are small epoch instances that are repeated multiple times back-to-back. These instances can be reliably measured because their repetition

[2]IPC and traffic volume can be measured in an interval since their min and max absolute values are known.

Parameter	*Value*	*Parameter*	*Value*
Processor model	in-order	L1 I/D Cache	
Issue width	2	Line size	64 B
L2 Cache		Size/Associativity	8 KB, direct-map
Line size	64 B	Load-to-Use latency	2 cycles
Size/Associativity	128 KB, 8-way	Network on Chip	
Tag latency	2 cycles	Topology	4×4 2D mesh
Data latency	6 cycles	Hop latency	3 cycles
Replacement policy	LRU	Main mem. latency	300 cycles

Table 3: Simulated machine architecture configuration.

creates a robust microarchitectural state. All other static epochs and their dynamic instances are included, independently of their granularity.

Experimental Setup

For various measurements we employ an elaborate CMP simulator built on Simics [86]. Our simulator is a detailed timing simulator, which models a 16-core tiled CMP with a 4×4 2D mesh network-on-chip (NoC), similar to the machine models used in recent studies and commercial developments [21, 115, 44]. Each tile of the processor incorporates a compute core, an L2 cache bank, coherence support, and a NoC router. A compute core is a two-issue in-order processor and has private L1 caches. Per-tile L2 caches form a globally shared logical L2 cache. Cache coherence is maintained by a distributed directory-based coherence protocol with MESI states, modeled after [74]. Table 3 summarizes our architecture configuration parameters.

We collect information for the programs listed in Table 2. For compilation, we use `gcc` version 4.2.0 with the `-O3` optimization level and the SunOS pthread library version 5.10. All programs were executed from start to finish with the problem sizes shown in Table 2, and referenced by [119, 13]. Statistics were collected starting from the beginning of the parallel section until the end. We use all available processor cores by spawning 16 threads in all experiments. Each thread was bound to a specific core for stable and repeatable measurements.

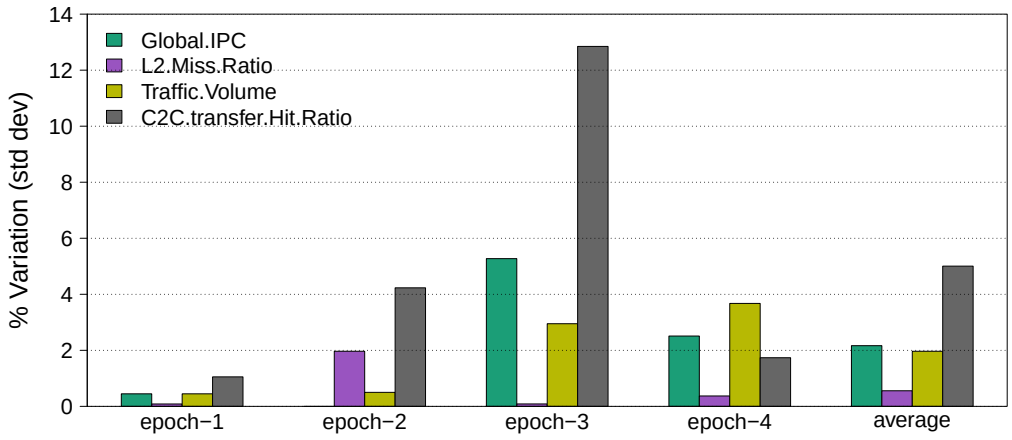

Figure 5: *Program behavior variation across each epoch's dynamic instances for bodytrack*: The variation measures the standard deviation across the instances of each epoch.

2.4.3 Variation Across Epoch's Instances

To characterize an epoch's behavior as a reoccurring pattern, we measure the variation of four different architecture metrics across the dynamic instances of each epoch. Figure 5 plots this variation for all four epochs of *bodytrack*.

The results show low variability for all the metrics across the instances of each static epoch. The percentages can directly evaluate the *similarity* or *stability* across an epoch's dynamic instances for the specific metric. For example, IPC varies around 2% across the instances of epoch-4. Assuming that 2% is a negligible variation, those instances can be considered similar to each other. Epochs with low variation for all the metrics reflect an epoch with a stable dynamic behavior. Comparing the variation between different epochs gives their relative stability. For example, the performance variation across the instances of epoch-3 is larger than that of epoch-1. Relatively higher variation is observed for C2C-transfer hit ratios, mainly because this metric is strongly affected by the prior to the barrier behavior, i.e., the cache state the previous epoch left. This preceding state could be different for each instance of an epoch. The best performance stability is demonstrated in epoch-1, which is the time-dominant epoch in *bodytrack*.

To extend our observations to all the other programs, we summarize in Figure 6 the average variation that is observed among the dynamic instances of each epoch, for each program. The

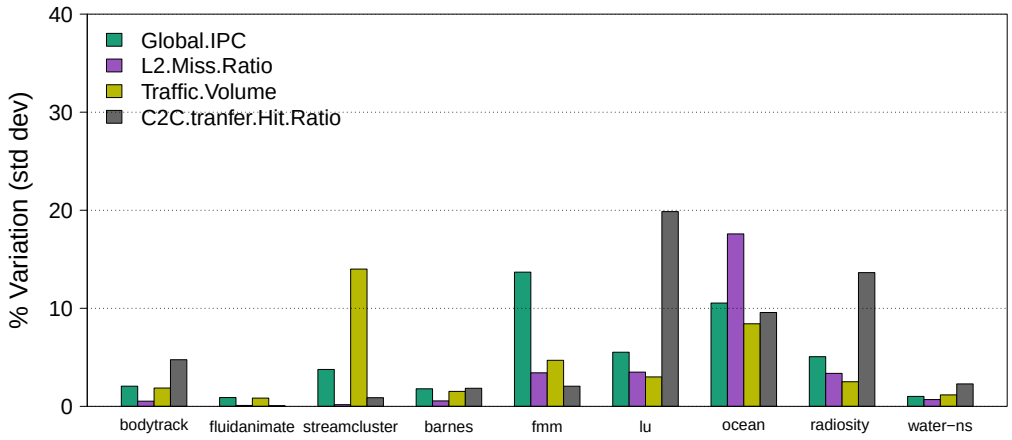

Figure 6: *Program behavior variation across each epoch's dynamic instances*: The lower the variation, the more effective is the epoch granularity in recognizing the cyclic patterns in program behavior.

average variation is weighted by the number of the dynamic instances of each epoch.[3] As the figure shows, most programs have very limited variation for all the metrics, highlighting the effectiveness of epochs defining periodic execution intervals that exhibit similar program behavior.

2.4.4 Variation Across Different Epochs

Figure 7 shows the variability in behavior among different epochs of *bodytrack*, as measured by each metric. In this example, each point reports the mean behavior of each epoch across its different instances. The error bars indicate the variation among the instances of the epoch, based on the variation analysis in the previous subsection.

The existence of high behavioral variability across different epochs (compared with low variability across instances of an epoch) indicates a fundamental correlation between epoch boundaries and changes in program behavior. A single metric that can capture this property (e.g., global IPC in the example of Figure 7) is adequate to show that there is such correlation. However, showing more than one metric is desirable since the same single metric may not be suitable for capturing the behavior changes in every application, even if such a metric exists (e.g., L2 miss ratio is fairly

[3] Valuable only from a statistical point of view; epochs with many instances are more unbiased estimators of the variation than epochs with fewer instances.

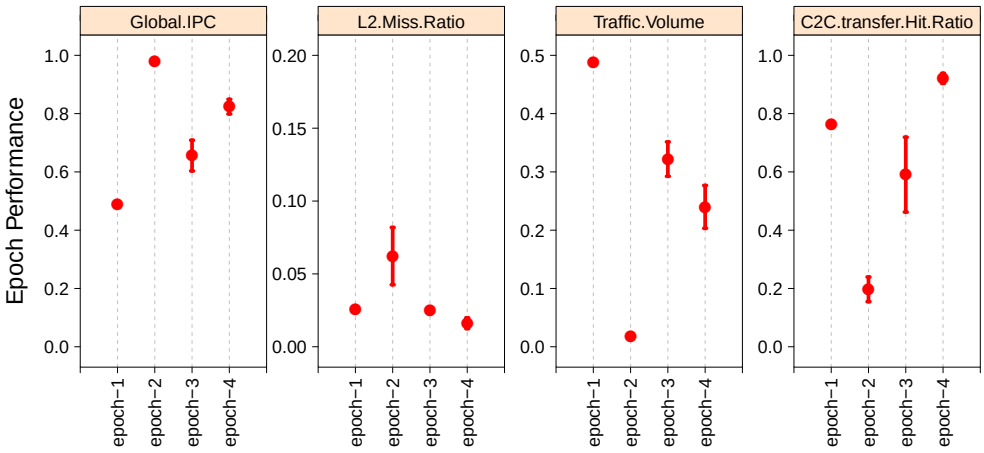

Figure 7: *Epoch-level behavior of bodytrack*: The behavior across different epochs varies significantly, compared with the variation that exists among epoch instances (denoted with error bars).

insensitive in the same example). In addition, the sensitivity analysis over a number of metrics against different epochs provides useful insights into how possible epoch-based optimizations can be directed and implemented on a target architecture. For example, an adaptive system can exploit the large variability of a metric across different epochs to improve the performance or power consumption of individual epochs.

To quantify how different epochs exhibit distinguishable behavior, we need to meaningfully compare the heterogeneity that exists across different epochs with the homogeneity that exists across instances of the same epoch. To measure this quantity, we use the ratio between the weighted average of the variation across the instances of each epoch (see error bars in Figure 7), and the variation that exists across the different epochs. The last part calculates the variation between the mean values taken from each epoch, compared with the grand mean of all the epochs' instances of the program. This ratio is lined to a statistical metric that is widely used in statistical analysis of variance (ANOVA), which calculates the significance of the variation between populations that are likely to have variation similarities within them [45]. Figure 8 shows the results for all the programs, and for each metric. If the behavior change across different epochs is significantly more marked than across their instances, we should see, at least for some metrics, a small ratio between

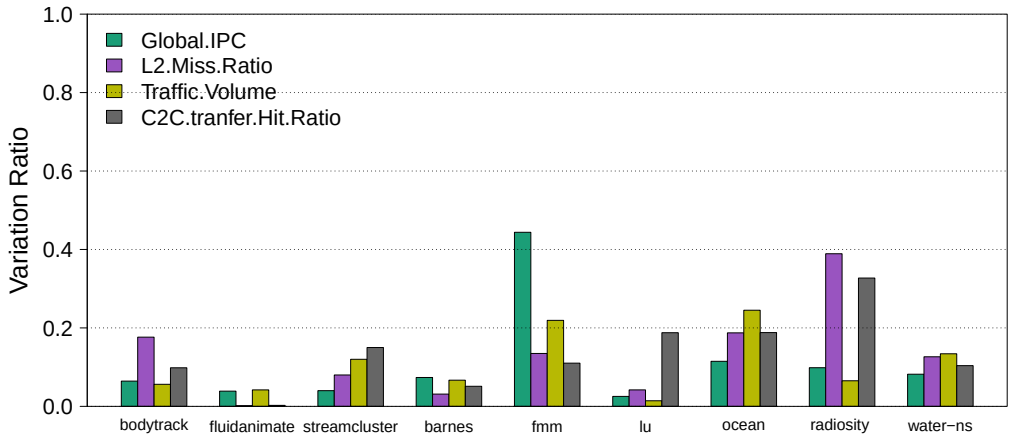

Figure 8: *Ratio between the variation of program behavior across epoch instances and the variation across different epochs*: The smaller the ratio, the sharper the behavioral shifts on epoch boundaries.

those standard deviations. The smaller the ratio, the better the distinction is.

The results show that all the examined parallel programs have at least some metrics with a very small ratio, indicating the power of epochs to distinguish the program behavior into well-defined intervals. The relatively high ratio in *fmm*, *radiosity*, and *water-ns* comes from the fact that their epochs are generally more similar than in other programs and not due to high variability among their epochs' instances (which is shown to be low in Figure 6).

2.4.5 Program Behavior Within Epochs

A program may exhibit further repetitive and shifting behaviors within epochs. For example, while most epochs in *bodytrack* have stable behavior, the L2 cache miss rate in epoch(C,D) (recall Figure 1) oscillates rapidly and as a result produces a highly "unstable" epoch. To gain more insight into the program behavior within epochs, we evaluate the relative stability or instability within their boundaries.

To perform our analysis, first we sample the behavior of each epoch using the four performance metrics. Sampling is done using fixed-length time intervals (250K cycles). Then, we split the epoch into *phases*[4], formed from consecutive intervals having stable behavior. To characterize

[4]Here we follow the traditional notion of program phase, defined as a collection of consecutive intervals

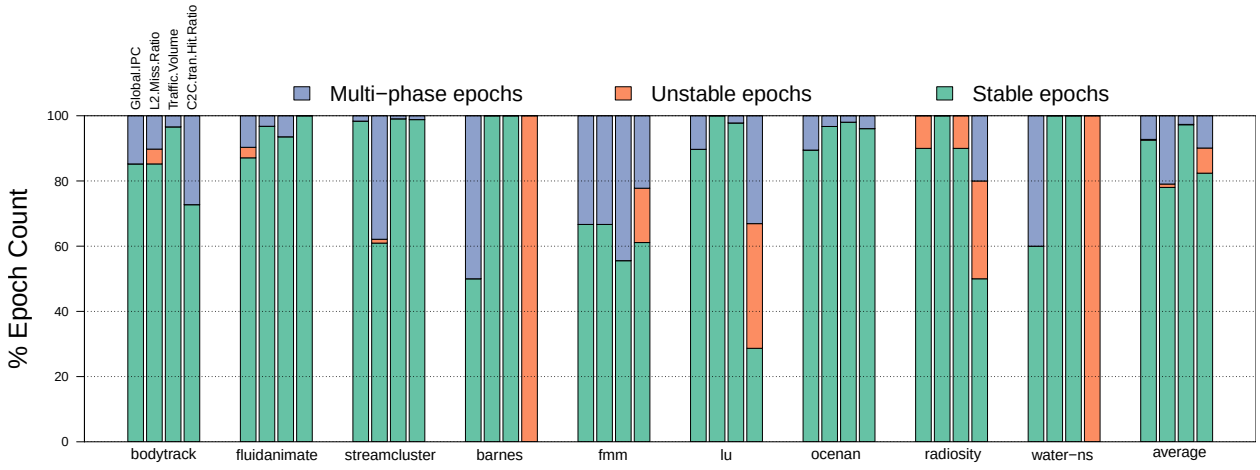

Figure 9: *Program Behavior within epochs (breakdown)*: Epochs show stable, unstable, or multi-phase behavior.

the behavioral stability within the epoch, we measure the length of each phase as well as the average phase length of the epoch. Based on those measurements, we classify each epoch as *stable*, *unstable* or *multi-phase*. A stable epoch has a single relatively large phase (our criterion is >80% of the epoch length) whereas unstable epochs have frequent changes of program behavior, and therefore will have many short phases. Consequently, we determine an unstable epoch if it has a short average phase length (<5% of the total epoch length). The rest are classified as multi-phase epochs, expected to have more than a single stable phase.

Figure 9 presents the result of our classification using the above criteria. On average, more than 80% of all epoch instances have stable behavior. Some epoch instances show unstable C2C-transfer hit ratio, mainly because this metric is more sensitive to thread interference and fine-grain communication patterns.

Although the behavioral changes within an epoch are not automatically inferred at the epoch granularity, the stability or instability, as consistent repetitive behavioral patterns, can be highly predictable. For example, consecutive instances of epoch(C,D) in *bodytrack* can be correctly predicted as unstable. We have observed that most program epochs have a strong tendency to repeat their internal pattern in their dynamic instances. Few exceptions were observed in epoch instances

with no significant performance changes (set to <5%).

having initialization parts or other relatively short epochs. Note that labeling an epoch as "unstable" in a coarser sense could have a much higher benefit than tracking rapid changes of very short regions. For example, attempting to continuously adapt and reconfigure a system within unstable regions can lead to unpredictable and undesirable results. In contrast, associating unstable epochs with appropriate system configurations will provide more effective overall impact.

2.4.6 Characterization Summary

Our epoch characterization shows that: (1) Epochs define intervals that repeat in a consistent and predictable way and therefore they provide a reliable granularity in which the cyclic pattern of program behavior can be observed; (2) Different epochs tend to have different behavior and therefore they provide an attractive granularity in which the program can be characterized. Consequently, epoch boundaries are likely to naturally indicate changes of program behavior; and (3) Most epochs exhibit stable behavior within their boundaries. In general, internal behavior patterns reoccur and thus can be accurately predicted.

2.5 TRACKING EPOCHS AT RUN-TIME

A system capable of tracking changes of program behavior at run-time can trigger a search for an optimal configuration that will adapt the system to a performance and/or power optimal state. The adaptation can be achieved through configurable hardware mechanisms [8, 28, 106, 57, 53]. Using an epoch-based approach, a system can dynamically detect a change of program behavior by "watching" for a barrier point. Then a search for the best configuration can be triggered to create and store a decision signature for the running epoch. Upon new iterations, the system could use the epoch ID to automatically reinstate the optimal configuration using the decision signatures stored by the previously seen epochs.

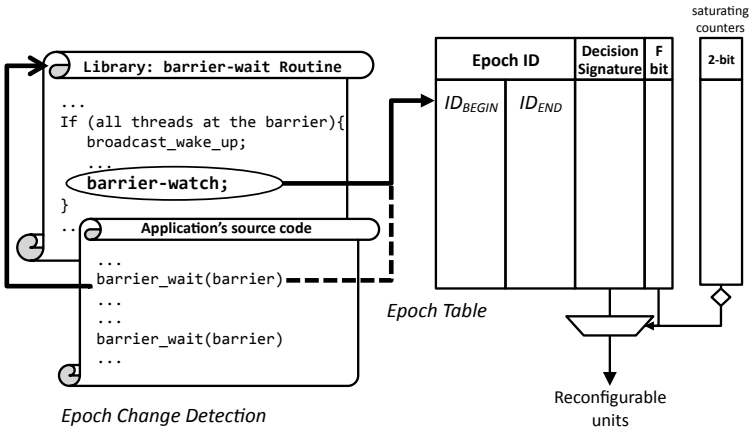

Figure 10: *Run-time epoch detection and epoch-based dynamic adaptation*: The BarrierWatch technique detects epoch transitions by capturing the barrier points. The epoch table associates epochs with decision signatures to be used for adaptive optimization.

2.5.1 Epoch Change Detection

A pass from one epoch to the next occurs when all the executing threads of a running program have reached a barrier. We expose this global synchronization point to the hardware through a special instruction, which is executed whenever a barrier is released. The "`barrier-watch`" instruction is located in the barrier synchronization routine and is tractable only by the last thread that reaches the barrier (Figure 10). The new library containing the `barrier-watch` instruction should be statically or dynamically linked with the application binary. Note that no modifications are required to the source code of the application. Whenever the `barrier-watch` instruction is executed, a barrier identity ($\text{ID}_{Barrier}$), which is unique for each static barrier call, is passed as a search key to the *epoch table*. The barrier identity is the return address of the barrier routine (obtained from the top of the call stack). Unless barrier calls are enclosed in wrapper routines, this address ($\text{ID}_{Barrier}$) is unique for each barrier point of the application.[5]

[5]In case the barrier calls are enclosed in wrapper routines, a unique address can be obtained by looking deeper into the call stack.

2.5.2 Epoch Table

The epoch table is a fully associative array which keeps the epoch IDs of the already executed epochs along with a decision mask and a filter bit for each ID. It can be implemented either in software as a routine linked with the program binary, or in hardware for fast adaptation decisions. A modest number of entries are sufficient to accommodate all the static epochs of a program (10 on average and 24 in the worst case for the evaluated programs). The ID field holds a tuple of two 64-bit barrier IDs (ID_{BEGIN} and ID_{END}) that correspond to the $\text{ID}_{Barrier}$ at the entry and exit point of the epoch. On a lookup operation, the search key is compared with each ID_{BEGIN} of the table for a match. If the entry is found, the *decision signature*, which is assumed to keep the optimal configuration for that epoch, will become available. The system can then use the decision signature to configure the hardware as directed. The use of the filter bit is to prevent, when necessary, the extraction of the decision mask. This might be desirable in several cases. For example, the decision mask might not exist or might not yet have reached a confident stage yet.[6] In another case, discussed further in Section 2.5.4, the filter bit suppresses the detection of very small epochs.

At the first instance of each epoch, no matching record is found in the table. This "compulsory miss" will allocate a new table entry, store the $\text{ID}_{Barrier}$ to the ID_{BEGIN} field and trigger an external process that will enable some kind of monitoring policy for the running epoch (e.g., enabling some hardware counters). At the end of the epoch, when the next barrier is reached and before the new lookup operation is performed, the new $\text{ID}_{Barrier}$ is stored in the ID_{END} field and the execution is transferred to a decision algorithm that evaluates the monitoring results, decides the best configuration, and saves the decision signature in the table entry.

The reason for keeping both ID_{BEGIN} and ID_{END} in the ID field is two-fold. The first has to do with the possibility of having two different epochs with the same ID_{BEGIN}. In that case, the ID_{END} is required to identify a false positive match. The second is that knowing the ID_{END} of the current epoch allows us to predict the next epoch in the future. Assuming that the current epoch ID is detected (i.e., the ID_{BEGIN} is known), a lookup at the same entry of the table can retrieve the ID_{END}, which is the ID_{BEGIN} of the following epoch.

[6] Depending on the adaptation approach, a decision algorithm may need more than one iteration to determine the signature.

Two different epochs can have the same ID_{BEGIN} if a control point can lead the execution path into different barriers during different iterations. Such an example is illustrated in Figure 3, where a branch after barrier C redirects the execution to a previous point multiple times, before proceeding. In this case, epoch(C,B) and epoch(C,D) have the same ID_{BEGIN}. Such a case happens if an application uses barriers at loop iteration boundaries.

Assuming that epoch(C,B) is recorded in the epoch table, epoch(C,D) will be incorrectly identified as epoch(C,B) during its first execution. To identify a false positive match, the ID_{END} is compared with the $ID_{Barrier}$ when the epoch reaches its exit point. The first false positive match of the (few) epochs like epoch(B,B) is inevitable and a new table entry will be allocated to accommodate the new epoch, as if there was no match. The new entry will have the same ID_{BEGIN} but a different ID_{END}. The filter bit might need to remain "set" until more iterations of the epoch are monitored and the appropriate decision signature is determined.

The possibility of having more than one epoch with the same ID_{BEGIN} in the epoch table will result in lookups with multiple matches. To avoid new false positives, the epoch detection mechanisms must be able to accurately predict and pick the entry with the correct epoch ID. In other words, the mechanism must be able to predict the dynamic sequence (i.e., trace) of barrier points (see next subsection).

The size of the epoch table is determined by the number of static epochs of the program. This number is typically small mainly because programming tactics generally encourage balanced and limited use of global synchronization points. Therefore, a possible hardware implementation can offer fast table lookups with negligible area and power overheads. A table with 24 entries (worst case in our experiments) occupies less than 0.5KB of capacity, and can be integrated on chip. If more space is required, a simple displacement policy such as LRU is likely to perform well, since there may be epochs that are used only once.

2.5.3 Barrier-Point Trace Prediction

Given a barrier-point, the barrier-point trace predictor will attempt to predict the next barrier point in execution order, or a whole sequence of following barrier points. As already described, barrier-point trace prediction is necessary for identifying epochs that start with the same starting barrier

ID but can end in different barriers each time. Nevertheless, barrier-point trace prediction—and more general synchronization-point trace prediction schemes—could be found also useful in related research areas such as software coherence, transactional memory, future path prediction, and deadlock detection.

We evaluate a prediction scheme that works similar to a simple 2-bit branch predictor [76]. The motivation behind the use of this scheme comes from the fact that the reoccurring sequence of synchronization points and the possible paths are often caused by easy-to-predict branches. In fact, most of the prediction attempts deal with no more than two matching options, suggesting that just a single branch is usually responsible for the multiple options. In general, the predictor works on the premise that a "path" has changed only after two consecutive mispredictions.

To explore the prediction scheme, a 2-bit saturating counter is added in the epoch table. At a barrier point, if the $\text{ID}_{Barrier}$ matches the ID_{BEGIN} of more than one entries, then the entry with the higher counter is picked and the next barrier is predicted accordingly. When the next barrier is reached, the prediction is evaluated by looking for a match with the ID_{END} of the entries that were initially candidates. The corresponding counters will then increase or decrease accordingly, depending on whether the ID_{END} matches or not.

Table 4 shows the effectiveness of the prediction method. Overall, the 2-bit predictor significantly reduces the mis-detected epochs. There are plenty of prediction models that can be used to further improve the accuracy of epoch detection or handle better multi-match cases [30, 118]. However, fully exploring a large design space of such prediction mechanisms is beyond the scope of this work.

2.5.4 Avoiding Small Epochs

Adaptation actions during epochs that appear as "noise" between longer epochs may have no practical benefit. Therefore, it is desirable for the epoch detection mechanism to recognize them. A small epoch can be detected either by the external process, which is called during the first instance of the epoch, or by a dedicated hardware. In both cases, the detection involves a calculation of the time difference between the entry and exit point of the epoch and a comparison with a predefined threshold (e.g., 50K cycles). After an epoch is labeled as small, it can be avoided either by

BENCHMARK	# DYN. EPOCHS	# MIS-DETECTED EPOCHS	
		W/O PRED.	2-BIT PRED.
bodytrack	19	4	1
fluidanimate	39	0	0
streamcluster	7,569	2,139	4
barnes	8	0	0
fmm	33	1	1
lu	258	1	1
ocean	707	40	4
radiosity	17	2	2
water-ns	19	4	2

Table 4: *Epoch static-ID prediction effectiveness (barrier trace prediction)*: High prediction accuracy is possible using a 2-bit prediction scheme.

enabling the filter bit of the corresponding entry or by not saving it in the epoch table. Using the first approach, future instances of the same epoch will automatically be classified as small and no optimization action will be taken.

2.6 EPOCH-BASED RESOURCE MANAGEMENT: IMPROVING THE PERFORMANCE/POWER TRADE-OFF OF THE NOC

In this section, we evaluate the effectiveness of epoch-based adaptation in a CMP architecture with a case study. The adaptation aims to optimize the energy and performance trade-off using Dynamic Voltage/Frequency Scaling (DVFS) applied to the NoC routers [90]. DVFS at NoC routers extends the concept of per-core DVFS to per-router DVFS for congestion and power management. In this study, we consider a global NoC DVFS regulator, and therefore all the routers of the network comply to the same voltage/frequency setting at a given time. Although this strategy is less flexible than per-router DVFS, it is practically more feasible and requires much less design and implementation effort.

Changes in the NoC's clock frequency shift performance and power in opposite directions.

Decreasing the frequency leads to lower power consumption in the routers, but negatively affects their congestion and processing speed. In addition, operating the NoC in a lower frequency than the processor cores can potentially incur network overloading and result in significant performance penalties. Thus, careful control of the NoC frequency is needed to exploit the energy/performance trade-off.

We note that the purpose of this case study is not to propose a complete solution for the energy/performance management of the NoC; rather, we aim to demonstrate the applicability of our simple and low-cost approach in the context of dynamic adaptation. Fully exploring the design space and comparing against other adaptive techniques is beyond the scope of this work.

2.6.1 Epoch-based Adaptation

We present two different implementations of epoch-based adaptation. In the first one, off-line profiling determines the frequency/voltage setting best suited for each epoch of the application ("static scheme"). We perform a profile run for each NoC voltage/frequency setting to record the execution time (D_f) and energy consumed (E_f) by each epoch (all the combined dynamic instances of each epoch). Then, we pick, for each epoch, the frequency/voltage level f_x for which $E_{f_x} \times D_{f_x}$ is minimized, i.e., to hit the best trade-off between energy and performance. The selected frequency level for each epoch is then included in the binary of the application as a decision signature. At run time and at the beginning of every epoch instance, the signature is retrieved and the NoC switches to the desirable frequency/voltage level.

In the second implementation, the epoch table and the decision signatures are determined dynamically at run time ("dynamic scheme"). A search for the best frequency level is triggered at the first instance of each epoch. During this period, all possible NoC voltage/frequency settings are examined by switching to a different frequency on a fixed time-interval basis within the epoch instance. Because the energy and performance measurements for each frequency have to be taken from different time intervals, we measure energy per instruction (EPI) and cycles per instruction (CPI). Thus, the best frequency f_x for each epoch becomes the one that minimizes $CPI_{f_x} \times EPI_{f_x}$. The selected frequency is then stored in the epoch table as a decision signature as described in Section 2.5. Upon new iterations of the same epoch, the signature is retrieved to adapt the system

CAPTION	FREQUENCY	VOLTAGE
$f_{100\%}$	3 GHz	0.8 V
$f_{75\%}$	2.25 GHz	0.65 V
$f_{50\%}$	1.5 GHz	0.5 V
$f_{25\%}$	0.75 GHz	0.35V

Table 5: Frequency/voltage levels. The linear relation between frequency and voltage is consistent with estimates in [66, 40].

accordingly.

The static, profile-based approach is particularly suitable to embedded and special-purpose environments where the execution environment is known *a priori*. Clearly, the static scheme is more effective than the dynamic scheme since optimal decisions will be readily available at run time. However, the dynamic scheme is more general and is applicable to a wider range of environments. We note here that even in dynamic environments, users often tend to execute a limited number of applications frequently, possibly solving similar problems repeatedly. In this case, the user (or the OS) can choose to store the applications' epoch tables to a persistent storage when they finish execution and pre-load them when the applications run again. Such a strategy can further improve the effectiveness of the epoch-based adaptation since all compulsory misses in the epoch table will be eliminated.

2.6.2 Evaluation Methodology

The experiments are performed using the same simulation environment as described in Section 2.4.2. In our simulator, the NoC models a wormhole-switched network with deterministic X-Y routing and ACK/NACK flow control. Data packets consist of six 128-bit flits and control messages one flit. Each router models a two-stage router pipeline and has 5 physical channels (PCs) and 2 Virtual Channels (VCs) multiplexed on each PC. We use a buffer size of 2 flits per VC.

For NoC DVFS, we assume four possible clock frequency and voltage levels, as shown in Table 5. $f_{100\%}$ represents the maximum operating frequency of the NoC, which we consider as the baseline frequency (no energy savings). DVFS policies are triggered during epoch transitions;

assuming on-chip voltage regulators, we measure 100 cycles for switching the voltage level.[7] For the dynamic implementation, we use 100k sampling intervals and thus the per-epoch monitoring period lasts at least 500K cycles (100k for warm-up and 100k per frequency level for at least one sample). During this period, we keep high voltage and we switch only frequency, therefore we have zero switching overhead and the energy consumption for the corresponding frequency level is estimated. Epoch instances are usually larger than 500k cycles, so a large number of samples are taken per frequency. When the first instance is smaller than 500k, the epoch is considered too short and is marked with the filter bit; later instances of the same epoch will therefore be skipped by the epoch detection mechanism.

Power consumption is modeled with dynamic, leakage, and background components. Dynamic power scales with fV^2 and leakage power with V. Background power, representing the cores and the rest of the system, is not in the same clock domain and consumes a constant amount of power. Leakage power consumption is assumed to be twice that of dynamic power when operating at 1GHz, which is consistent with estimates from Kim et al. [66]. Background power is computed assuming that a loaded NoC at 1.5 GHz consumes 20% of the total system power [65].

2.6.3 Evaluation Results

Figures 11, 12, and 13 compare energy savings, execution slowdowns and energy-delay improvement, respectively, between non-adaptive fixed frequency schemes and epoch-based adaptation schemes, for the reference applications. Results are shown with respect to the baseline scheme, where the NoC operates at full frequency $f_{100\%}$.

As the NoC frequency decreases, the power consumed in the NoC is reduced and the overall on-chip energy savings are expected to grow. On the other hand, when NoC operates in lower frequencies, more time is needed for the application to complete execution, making unclear whether the benefits gained by the low-power NoC will result in system-wide energy savings at run completion. For example, the results show that when the NoC operates at $f_{25\%}$, the slowdown is large enough to adverse the NoC energy savings into significant sytem energy loss. The same effect is also observed occasionally for other sub-optimal fixed frequencies. On average, all schemes except

[7] State-of-the-art on-chip voltage regulators can transition voltage even faster (5ns) [40].

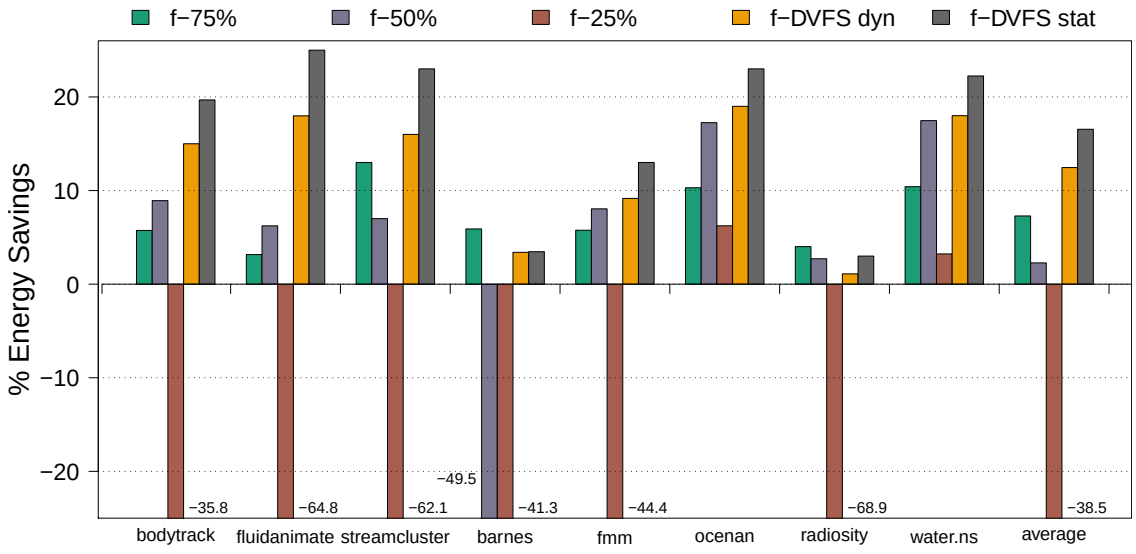

Figure 11: *Energy savings from Energy-Delay NoC management:* Epoch-based NoC DVFS is compared to static VF schemes. Results are normalized to the baseline static scheme $f_{100\%}$).

$f_{25\%}$ save energy at the cost of performance. Note that in general, the overall energy savings are directly affected by the portion of the on-chip power consumed by the NoC.

Our results demonstrate the effectiveness of both static and dynamic epoch-based DVFS schemes in adapting the system into a more efficient state. In all cases, both adaptive schemes show energy reductions for the least performance degradation compared with the fixed schemes. The static off-line scheme achieves about 16.6% energy savings on average, with 2.7% slowdown. The savings achieved by the dynamic on-line scheme for roughly the same average slowdown are about 12.5%, which is around 75% of what the off-line scheme achieves. In general, as Figure 13 shows, the dynamic scheme follows closely and consistently the effectiveness of the static scheme, indicating the strength of the dynamic approach in capturing the changes in program behavior correctly. In the case of *barnes* and *radiosity*, where a single epoch is dominating the execution, a wrong voltage/frequency adaptation decision for that epoch could significantly impact the results. Based on the ED results, both approaches reach the same decision and successfully pick the optimal frequency.

Overall, our evaluation shows that the BarrierWatch approach is robust in detecting phase

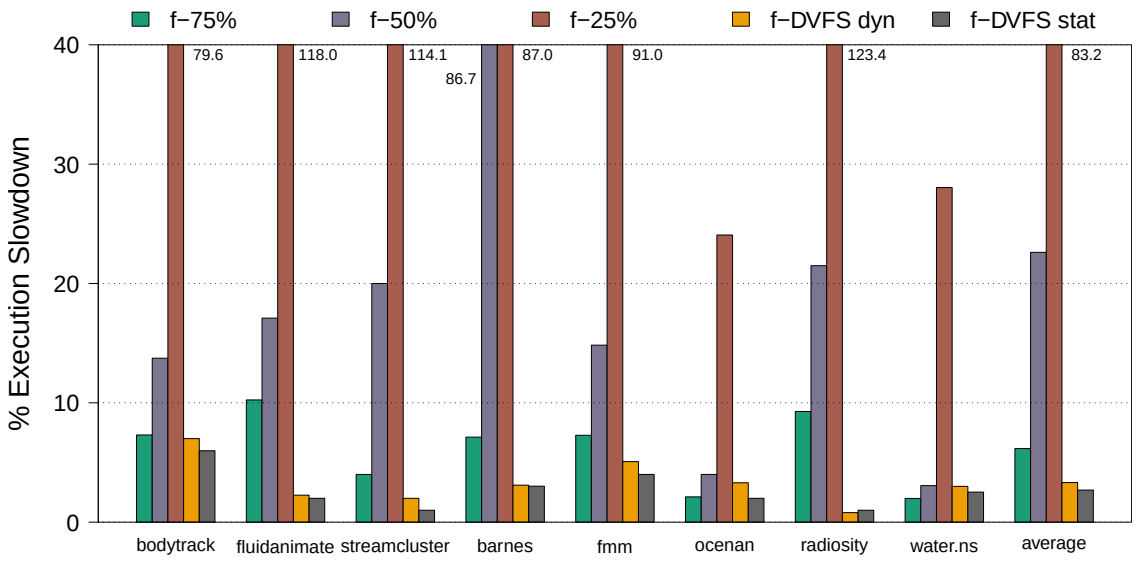

Figure 12: *Execution slowdown from Energy-Delay NoC management:* Epoch-based NoC DVFS is compared to static VF schemes. Results are normalized to the baseline static scheme $f_{100\%}$).

changes and adapting an epoch-aware system effectively. The lightweight detection of a phase change using epoch boundaries is limited, however, to the epoch granularity. Further techniques are required if changes within large multi-phase epochs need to be handled. Apart from accurately detecting behavior shifts, the effectiveness of an epoch-based adaptation strategy depends on the accuracy of the decision signatures and the underlying configurable hardware.

2.7 SUMMARY

This work presents a characterization study of multithreaded workload behavior across and within epochs. Epochs are variable-length execution intervals defined by the global synchronization points that exist in multithreaded applications. Being program-defined and globally visible to all running threads, epochs are a natural granularity for examining the time-varying behavior of multithreaded workloads. Our analysis reveals that epochs repeat with strong behavioral similarity, while their boundaries indicate a significant behavioral change. Based on this observation, we pro-

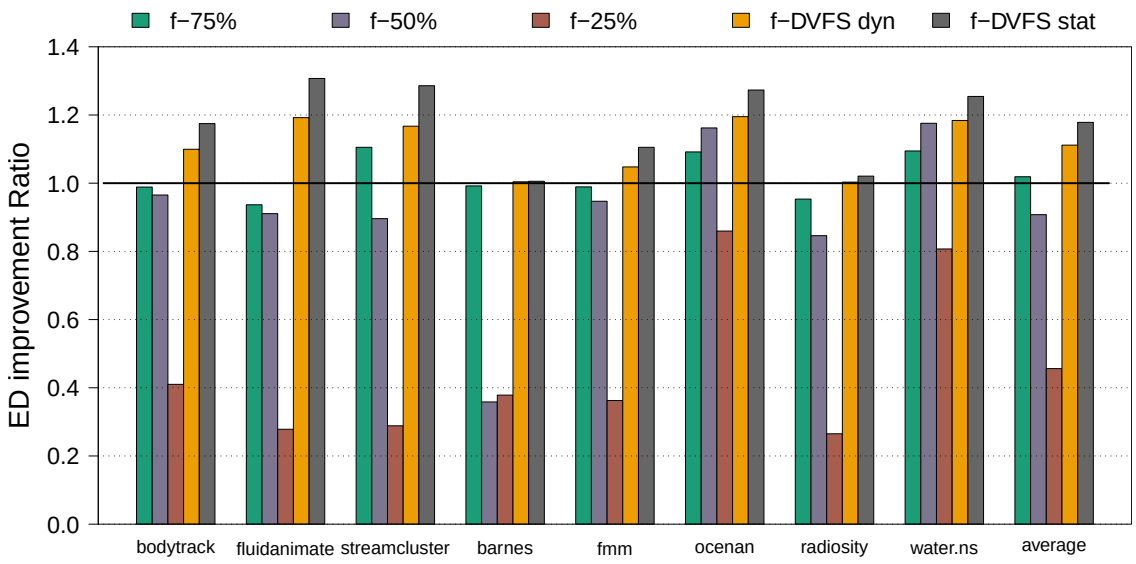

Figure 13: *Overall Energy-Delay NoC improvement:* Epoch-based NoC DVFS is compared to static VF schemes. Results are normalized to the baseline static scheme $f_{100\%}$).

pose an effective and lightweight technique for dynamically detecting and predicting changes in program behavior with the epoch granularity. Desirable properties of our approach include: being independent from the underlying architecture, requiring no monitoring of fine-grained fixed execution intervals, naturally adopting variable-length intervals, being amenable for low-cost implementation and deployment, and finally being applicable to many multithreaded workloads written with barrier synchronizations.

Detecting changes in program behavior is essential for adaptive program optimization, both static and dynamic. Watching for synchronization points provides a simple and elegant approach to capturing certain variations in program behavior. The proposed framework opens new opportunities for run-time optimizations and effective resource management in future CMPs.

3.0 COHERENCE COMMUNICATION PREDICTION

3.1 INTRODUCTION

Inter-thread communication in shared memory systems is realized by allowing different threads to access a common memory space. This model simplifies the concept of communication; however, it creates important scaling challenges mainly due to the cache coherence problem [74]. Traditionally, shared memory architectures employ either a directory- or a snooping-based protocol to keep the per-processor caches coherent. Directories maintain a full sharing state for each cache line and therefore can precisely direct each miss to its destinations. The indirection to the directory adds, however, considerable extra latency to cache misses that are serviced by other caches. Snooping protocols avoid the latency and storage overheads of a directory by resorting to broadcasting messages on each miss; however, they place significant bandwidth demands on the interconnect even for a moderate number of processors.

A common approach to improving coherence communication is to predict the processors to which a coherence request must be delivered to. Accurate prediction would reduce the latency of a cache miss by avoiding indirection to the directory, or reduce the high bandwidth demands of broadcasting by using multicasting in snooping protocols. Such predictions can be made by programmers (e.g., [1]), compilers [67, 116, 103, 81], or transparently by the hardware [75, 92, 69, 61, 15, 70, 2, 3, 88, 20]. Given that compiler techniques are limited to static optimization [67] and that the shared memory model should be kept transparent while offering high performance [92], a preferred communication predictor would dynamically learn and adapt to an application's sharing behavior and communication patterns.

Much prior work explored coherence target prediction using address- and instruction-based approaches [92, 61, 69, 15, 62, 2, 3, 88]. Address-based coherence prediction was first proposed

by Mukherjee and Hill [92], who showed that coherence events are correlated with the referenced address of a request. To exploit the correlation, they associate pattern history tables with memory addresses, train them by monitoring coherence activity, and probe them on each request to obtain prediction. Alternatively, instruction-based prediction, as proposed by Kaxiras and Goodman [61], correlates coherence events with the history of load and store instructions. This allows a more concise representation of history information since the number of static loads and stores is significantly smaller than that of accessed memory blocks.

The basic design of address- and instruction-based predictor has been extended further to mainly relax the large space requirements of those approaches [69, 15, 88, 93]. However, the extensions still require relatively large and frequently accessed prediction tables. Furthermore, to attain high accuracy, they often keep long sharing pattern history per entry or rely on multi-level prediction mechanisms. Designs that exploit the spatial locality of coherence requests, such as the ones based on macroblock indexing [88], have shown improvements for both space efficiency and prediction accuracy, indicating that predicting sharing patterns at very fine granularities is not necessarily optimal. Nevertheless, the window for capturing such opportunities is still tied to hardware-level observation, limiting the scope in which communication localities can be expressed and exploited.

In this work, we propose *Synchronization Point based Prediction* (SP-prediction), a novel runtime technique to predict coherence request targets. SP-prediction builds on the intuition that *inter-thread communication caused by coherence transactions is directly related with the synchronization points* in parallel execution. The main idea of SP-prediction is to dynamically track communication behavior across synchronization points and uncover important communication patterns. Discovered communication patterns are then associated with each synchronization point in the instruction stream and used to predict the communication of requests that follow each synchronization point.

SP-prediction is different than existing hardware techniques because it exploits inherent application characteristics to predict communication patterns. In contrast to address- and instruction-based approaches, it associates communication patterns with *variable-length, application-defined execution intervals*. It also employs a simple history structure to recall past communication patterns when the program execution repeats previously seen synchronization points. These two properties

allow a very low implementation cost and hardware resource usage, yet delivering relatively high performance. In summary, this work makes the following contributions:

• We examine the communication behavior as observed between synchronization points for various multithreaded applications (Section 3.3). Our characterization reveals prominent prediction opportunities by identifying: (1) strong communication locality during periods between consecutive synchronization points; and (2) predictable communication patterns across repeating instances of such periods.

• We propose SP-prediction, a run-time technique to accurately predict the destination of each coherence request using a small amount of hardware resources. SP-prediction captures synchronization points at run time and monitors the communication activity between them. By doing so, it extracts a simple communication signature and uses it to predict the set of processors that are likely to satisfy coherence requests of the program interval, as well as requests that will occur in future dynamic instances of the same interval (Section 3.4).

• We fully evaluate SP-prediction over a directory-based coherence protocol on an elaborate chip multiprocessor (CMP) model (Section 3.5). Our results show that SP-prediction can accurately predict up to 75% of the misses that must communicate with other caches, without adding excessive bandwidth demands to the baseline directory protocol (below 10% of what broadcasting would add). Correct predictions translate into sizable reduction in miss latency (13% on average) and execution time (7% on average) compared to the baseline directory protocol. Compared to existing address- and instruction-based predictors, our approach achieves comparable performance, albeit at significantly lower cost.

3.2 BACKGROUND AND MOTIVATION

Communicating misses. Coherence communication occurs on every memory request that must contact at least one other processor in order to be satisfied. These requests, also known as *communicating misses*[1], are read/write misses or write upgrades (upgrade misses) on cache blocks that have valid copies residing in non-local caches. Prior studies have shown that many applications

[1]"Coherence request", "coherence miss", and "cache-to-cache miss" are also commonly used names.

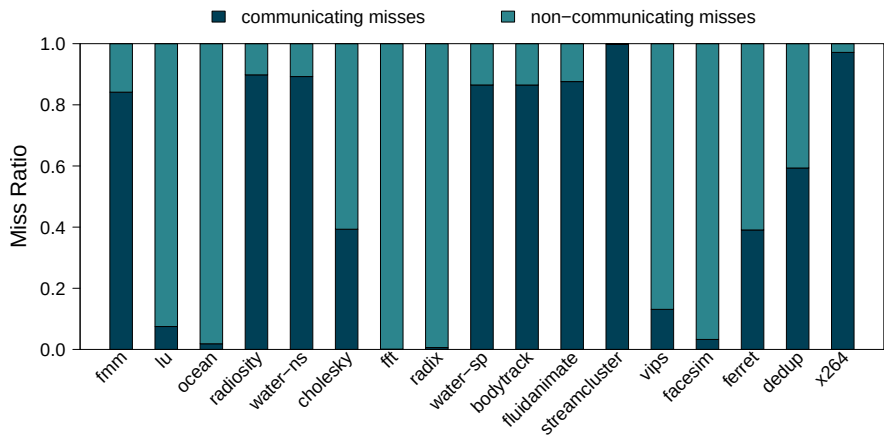

Figure 14: *Ratio of communicating misses.* (Note: Details on the evaluation environment are given in later sections.)

incur a large fraction of such communicating misses [10, 88]. This fraction depends primarily on application characteristics like working set size, data sharing, data reuse distance, and cache parameters. Figure 14 shows results for the workloads studied in this work. On average, communicating misses account for 62%, with considerable variation among different applications. In general, applications with a high rate of communicating misses benefit from coherence target prediction.

Coherence communication prediction. Predicting the communication requirements of a coherence request involves guessing a set of processors *sufficient* to satisfy a given miss. A prediction scheme may exploit the communication behavior of recent misses to predict the next one, assuming that misses exhibit temporal communication locality. For instance, a prior study has shown that the two most recent destinations grab a cumulative 65% chance of sourcing the data of the next miss [59]. Communication locality is better captured, however, if misses are tracked based on the address they refer to, or the corresponding static instructions, thus motivating the address- and instruction-based prediction approaches.

Address-based prediction builds on the expectation that misses to the same address (cache block) will have to communicate with the processor that wrote on the same address previously, or the set of processors that read from the same address recently. Tracking misses in such fine granularity, however, adds significant area requirements. To reduce the overhead, a practical address-based predictor is implemented with limited capacity (i.e., as a cache), and/or indexed by blocks

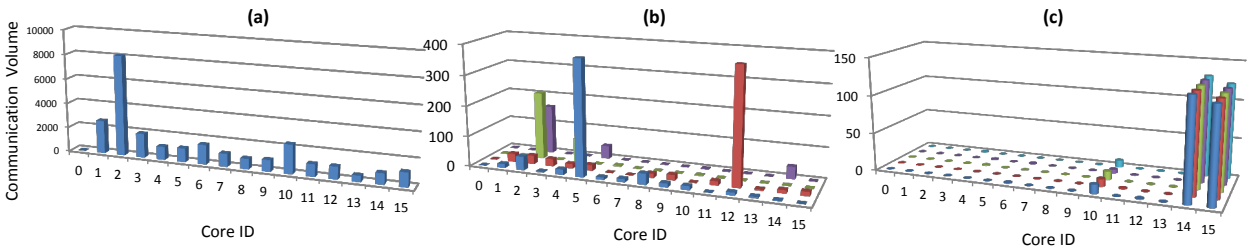

Figure 15: *Communication Distribution of Core 0 in bodytrack:* (a) As seen during the whole execution. (b) As seen during the execution of four consecutive synchronization-defined sub-intervals. (c) As seen across five different dynamic instances of the same sync-defined interval.

of larger granularity, e.g., a macroblock or page. As for macroblock indexing, it has in fact been shown to improve both accuracy and space efficiency, since misses on adjacent addresses are likely to have identical communication behavior [88]. Similar in concept and motivation, instruction-based prediction resorts to the expectation that misses generated by the same static instructions will have related coherence activity. This compacts further the tracked information since the number of static load and store instructions is much smaller than the number of data addresses accessed.

The above prediction approaches are typically implemented as hardware mechanisms that consume a considerable amount of resources and are unaware of any application-level characteristics. However, the way parallel applications are coded and structured embodies intuition to create high-level understanding of how communication activity occurs and changes through time. This work examines the idea of exploiting such opportunity through the synchronization points that exist in applications.

Synchronization points. The shared memory model eliminates the explicit software management of data exchange between processors. Nevertheless, race conditions between concurrent threads require the explicit enforcement of synchronization points, to ensure that operations on shared memory locations are consistent. As a result, although those points tell nothing specific about the inter-thread communication, they naturally indicate the points when certain data private to a processor will become visible—and possibly be communicated—to other processors. In what follows, we give a motivating example that shows how synchronization points partition the execution of an application into intervals, capture the existing communication locality in the application and, expose the repeatability of those partitioned intervals throughout the execution.

Figure 15 plots how a processing core communicates with other cores on a simulated 16-core CMP over: (a) the whole execution; (b) different execution intervals; and (c) dynamic instances of a single interval. By zooming into a granularity defined by synchronization points (plot (b)), it becomes clearer that the spatial behavior of the communication is strongly related to the specific intervals chosen. The sharp changes in communication behavior at the interval boundaries suggests that synchronization points are likely to indicate directly when behavior changes, and potentially hint a predictor to adapt faster to such changing behavior. In addition, the small set of processors that are contacted during each interval suggest that tracking the behavior on individual addresses or instructions within the interval may not necessarily result in a more accurate prediction. Lastly, predictable communication patterns that may appear across the dynamic instances of the same interval (plot (c)) create a new scope of temporal predictability and a key opportunity to exploit the repeatability of the communication behavior.

To illustrate how such variations in communication behavior are manifested through shared memory programming practices, we list a simple example code in the following. Shared data (`ME` and `LE`) are exchanged between parents, children, and siblings in a tree-like structure, which has its nodes arranged across multiple processors in a balanced way. During interval A, processors act as leafs and communicate data from processors where their parents and parents' sibling nodes reside. However, during interval B, processors act as inner nodes, hence the communication direction switches towards the set of processors that hold their children. This shift can be successfully detected and exposed by the synchronization point separating the two intervals.

Example Program Code

```
for nodes in this processor:
...
barrier();                          // interval A begins
node is a leaf:
  p = node.parent.LE[];
  for some node.parent.sibling:
    ps = node.parent.sibling.LE[];
...
barrier();                          // interval A ends
...                                 // interval B begins
node is a parent:
  for each node.child:
    node.LE[] = translate(node.child.ME[]);
...
barrier();                          // interval B ends
```

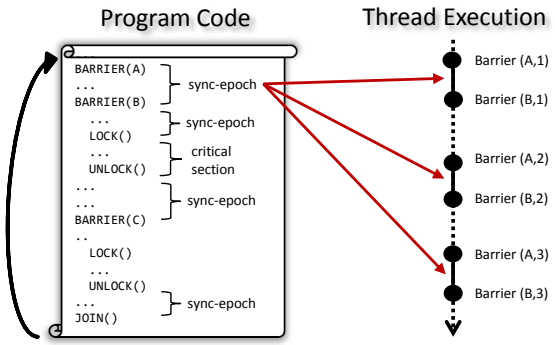

Figure 16: Static and dynamic sync-points and sync-epochs.

3.3 COMMUNICATION CHARACTERIZATION

The communication behavior of a core over a certain interval can be characterized by the target cores with which it communicates (called *communication set*) and the *distribution of the communication volume* across that set. We have already shown examples of such distributions in Figure 15. In this section, we first introduce simple notions about synchronization point based intervals, and then we characterize the communication behavior of those intervals for various workloads.

3.3.1 Synchronization based Epochs

The concept of synchronization point (*sync-points*) and synchronization epoch (*sync-epochs*) follows the description and definitions given in Sections 2.3.1 and 2.3.2. In this Chapter, a sync-epoch refers to an execution interval enclosed by two consecutive sync-points of any type. Therefore, a sync-epoch is defined at the thread-level, and is essentially a finer decomposition of the global-epochs examined in Section 2.3. Figure 16 depicts different thread-level sync-epochs and the notion of a static and dynamic ID.

BENCHMARK	# STATIC CRIT. SECT.	# STATIC SYNC-EPOCHS	PROGRAM INPUT	# TOTAL DYN. SYNC-EPOCHS
fmm	30	20	16K (particles)	2,789
lu	7	5	521 (matrix)	185
ocean	28	20	258 (grid)	2,685
radiosity	34	12	room	17,637
water-ns	20	8	512 (mol.)	1,224
cholesky	28	27	tk15.O	1,998
fft	8	8	256K (points)	22
radix	8	4	4M (keys)	35
water-sp	17	1	512 (mol.)	83
bodytrack	16	20	simsmall	456
fluidanimate	11	20	simsmall	8,991
streamcluster	1	24	simsmall	11,454
vips	14	8	simsmall	419
facesim	2	3	simsmall	3,826
ferret	4	6	simsmall	25
dedup	3	4	simsmall	508
x264	2	3	simsmall	56

Table 6: Sync-epoch statistics of benchmarks (per core average).

3.3.2 Simulation Environment

For the characterization study in this section, we employ a 16-core CMP model based on Simics full-system simulator [86]. The target system incorporates 2-issue in-order SPARC cores with 1MB private L2 cache, and a MESIF coherence protocol [55]. To track inter-core communication, we collected L2 miss traces that contain the miss data address, type, PC, and the target set of cores it must communicate with. The traces also contain all sync-points along with their type and static/dynamic IDs. Traces do not capture the effects of timing and are used only for characterization purposes. A full evaluation of our prediction scheme uses a detailed execution-driven performance model and is described in Section 3.5.

We study benchmarks from the splash2 and parsec suites [14, 119]. Table 6 lists key statistics related to sync-epochs for each studied benchmark. We use all available processor cores by spawning 16 concurrent threads in all experiments. For stable and repeatable measurements, we prevent thread migration by binding each thread to the first touched core. This was done except for dedup, ferret, and x264, because they create more threads than the available CPUs and rely on the OS for scheduling. Section 3.5.5 describes how our scheme can handle thread migration.

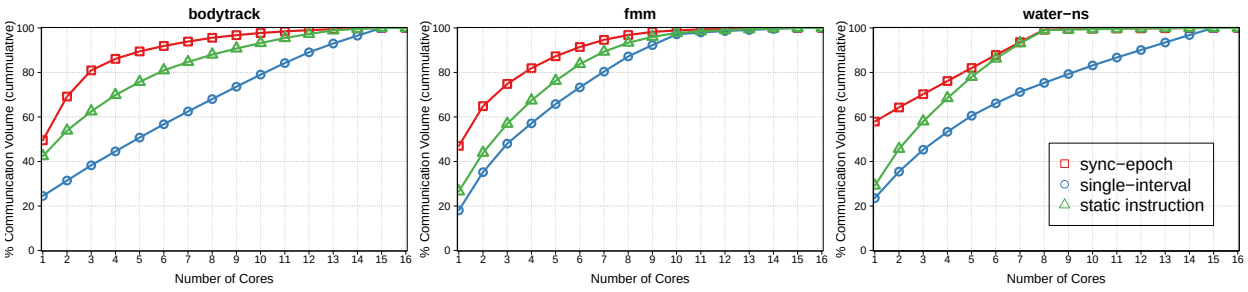

Figure 17: *Average communication locality of bodytrack, water-ns, and fmm:* Each curve shows the average cumulative communication distribution as seen in different granularities. Higher communication coverage for a given number of cores translates to better communication locality.

3.3.3 Communication Locality

The distribution of the communication volume characterizes the spatial behavior of the communication during an interval and illustrates whether it is "localized" to a certain set of targets. Examples of such localization are clearly observable in the communication distributions of Figure 15. For instance, core 0 during the first sync-epoch in example (b) communicates mostly with a single "hot" target, core 5, while nine other targets are contacted sporadically.

The communication locality is expressed by measuring the amount of communication volume that is *covered* by a certain number of cores. Using the previous example, core 5 covers more than 90% of the communication volume. Generally, if each individual miss communicates with C targets on average and the overall volume of the interval appears to be fully covered by C cores, then the interval has a perfect locality. When comparing different intervals with a similar C value, we can simply say that the locality is better when the communication is concentrated to fewer destinations.

A question that arises is *how good* is how good this locality is relative to various granularities. For example, based on Figure 15(a), one could argue that a certain level of locality also exists at the whole execution granularity since core 2 is "hotter" than the rest. To answer this question, Figure 17 shows the communication locality in applications, as captured by three different granularities: the sync-epoch granularity, the whole interval (as in Figure 15(a)), and the one that is based on static instruction indexing. Curves display average cumulative distributions over the

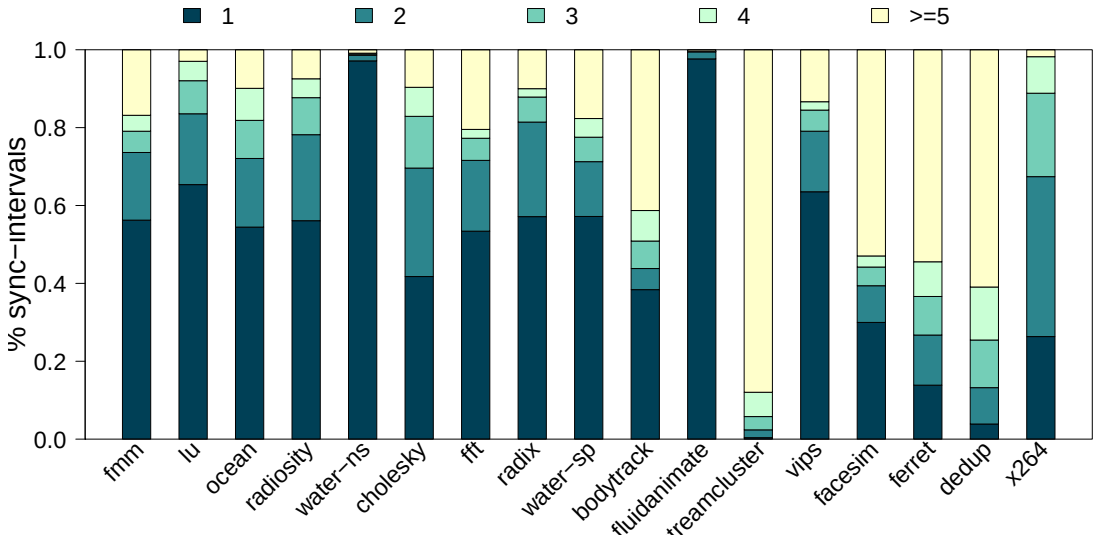

Figure 18: *Distribution of intervals based on their hot communication set size:* More than 78% of intervals have a hot communication set size smaller than or equal to 4.

whole execution and each point in the curve directly measures the average volume covered by a certain number of cores.

3.3.4 Dynamic Instances of Sync-Epochs

As the comparison shows, sync-epochs can capture the communication locality considerably better than a direct observation over the whole execution, suggesting that localities in communication's spatial behavior are closely related to sync-epochs. Moreover, sync-epochs often show better locality even to instruction-based granularity. This implies that communication activity could possibly be tracked as effectively as in traditional methods using sync-epochs, which is a much coarser-grain granularity. The results indicate that, overall, sync-epochs are attractive for extracting and exploiting repeatable communication behavior.

To create a representative signature of the communication behavior over each execution interval, we derive a *hot communication set* for each sync-epoch. A core is considered hot if it draws more than a certain amount of the total communication activity in the interval. Hence, the hot set could be formed based on a threshold over the communication distribution of the interval. The

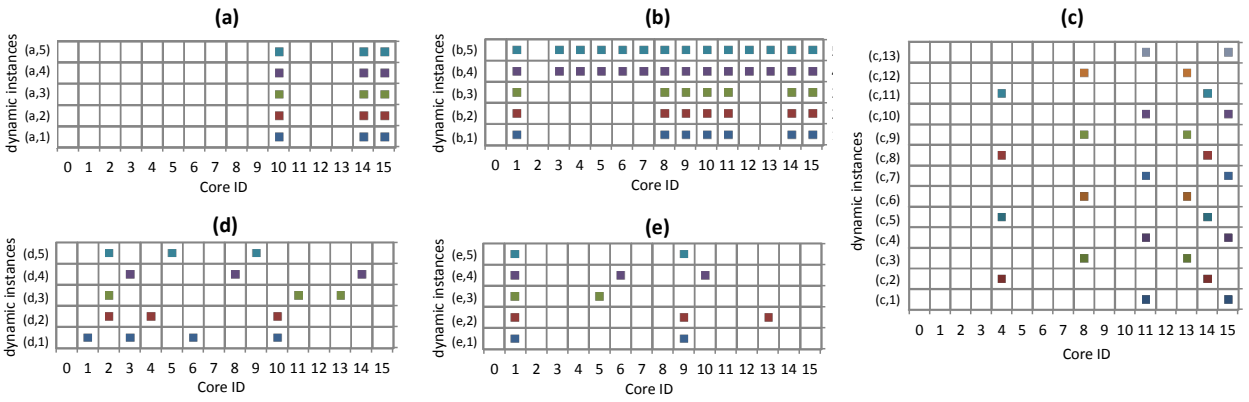

Figure 19: *Examples of hot communication set patterns across dynamic instances of a sync-epoch:* (a) Stable pattern. (b) Change from one stable pattern to another. (c) Repetitive pattern with stride 3. (d) Random pattern (critical section). (e) Combination of stable and random hot destinations.

size of the set represents the amount of the interval's hot targets. Figure 18 shows the distribution of sync-epochs for each application based on the size of their hot communication set. The results consider a threshold of 10%, meaning that a core is considered hot if it is contacted by at least 10% of the total communication activity of the interval. In contrast to Figure 17 where only the average number of the hot communication set size is clear, the latter figure shows how this size varies among the sync-epochs of the applications. Note that to further measure how close the hot set size is to the optimal locality, one should also consider the average communication set size per miss.

Sync-points are executed repeatedly and create a sequence of dynamic instances for each sync-epoch. As these instances exercise the same or similar code and operate on the same (or related) data structures, it is likely that they present behavioral similarities among themselves [26]. Such similarities or variations may also be reflected in the communication's behavior, depending on how the shared data are accessed in each instance, their sharing patterns, the level of determinism, and possible machine artifacts, e.g., local cache capacity and false sharing effects.

Here we present our general observations on how communication activities appear in the dynamic sync-epoch instances in the examined applications. Our findings are derived from extracting the hot communication set of every dynamic instance of a sync-epoch, and characterizing how it

changes from instance to instance.

Hot communication set patterns. Hot communication sets change across the dynamic instances of a sync-epoch following a predictable or random pattern. We categorize the patterns into: *Stable*, *repetitive*, *random*, or some *combination* of these. Figure 19 illustrates example patterns by representing each hot communication set as a bit vector.

Stable hot communication sets occur when the majority of the data consumed each time are provided by a single core. This case is common in applications with stable producer-consumer sharing aligned to sync-epoch granularity. Hot communication sets that follow repetitive patterns are commonly found in fairly structured parallel algorithms that exercise a different but finite number of data paths on different sync-epoch iterations. For similar reasons, communication sets may also demonstrate spatial-stride or next neighbor patterns. In contrast, random patterns are usually caused by accesses on migratory and widely shared data that are produced/consumed in a non-deterministic order. Those occur when threads repeatedly compete before they are granted the privilege to produce data that will be shared (e.g., accesses within critical sections), or when the data sharing sequences are dynamically determined by the parallel algorithm (e.g., decisions made within critical sections). Patterns that appear to combine various patterns are usually an artifact caused by the granularity in which we track the communication (e.g., a long sync-epoch may span across multiple functions and data structures, each having different sharing patterns).

"Noisy" sync-epoch instances. Often, some dynamic instances of a sync-epoch appear to have very low communication activity relative to other instances. This is usually caused by a control statement, which forces specific instances to flow through different execution paths that exercise code with relatively few accesses to shared data. Such instances may not give a representative sample when forming a hot communication set due to statistical bias; therefore, we treat them as noise and exclude them from the dynamic pattern.

3.4 SYNC-EPOCH BASED TARGET PREDICTION

The existence of communication locality at the sync-epoch granularity implies that misses within the sync-epoch are likely to communicate with processors in the hot communication set. Thus,

the hot set, if known, could be a *relatively small* and *sufficient* target predictor for the majority of misses within the interval. Based on this observation and, on evidence that many hot communication sets are predictable, we propose **SP-prediction**, a run-time scheme that exploits the temporal predictability within and across sync-epochs to predict the communication destination of misses.

SP-prediction is different from other prior approaches that exploit the temporal sharing patterns of misses in two fundamental ways. First, it makes use of the communication locality over application-specific execution intervals to predict for each miss in the interval, with no reliance on the temporal communication locality between consecutive misses. This is a significant advantage when communication locality is only seen among a broader temporal and spatial set of misses. Second, it can recall communication patterns from the past at a sync-epoch granularity and not for a specific address or instruction. This may allow the predictor to adapt quickly to old and forgotten patterns without complex mechanisms and long history information.

3.4.1 Basic Idea of Run-Time Prediction

SP-predictor exploits sync-epochs' communication locality to predict the destinations of a miss. Each program thread is seen as a sequence of sync-epochs, many of which are exercised multiple times during program execution. Obtaining a predictor of the communication behavior in a sync-epoch involves retrieving history information from previously executed instances of the same sync-epoch, as well as tracking the coherence communication of the currently executed interval. Each private L2 cache controller would hold the obtained predictor and accelerate miss-incurred communication by invoking a prediction action in the standard coherence protocol on each miss.

Synchronization primitives are exposed to the hardware so that it can identify the sync-epochs and sense their beginning and end. This requires simple annotations in the related software library (or program code) and corresponding support in the hardware. The hardware design cost entails the addition of a new instruction that retrieves the PC or lock address of the sync-point and forwards it to the coherence controller. The insertion of the instruction in the code is trivial and could be done by the library developer or automatically by a compiler. We consider that such support is feasible in today's hardware and software, and similar implementations exist (e.g., [19, 112]).

EVENT	ACTION
Sync-point captured (*sync-epoch begins*)	- Store sync-epoch's tag and type into SP-table. - Reset all communication counters.
Data response on RD/WR-miss	If the response comes from a remote node's cache: - Increment communication-counters[responder].
Invalidation Ack responses	- Increment communication-counters[responders]
Sync-point captured (*sync-epoch ends*)	- Extract hot communication set from counters - Store the hot set as a signature to the SP-table

Table 7: Building communication signatures.

3.4.2 Building Communication Signatures

Each processor monitors its communication activities by tracking responses to misses that have invoked the coherence protocol. A set of *communication counters* record the overall communication towards each destination. Responses for read misses include the data provider's ID and increment the communication counter that corresponds to the source processor. Responses for write and upgrade misses include a bit vector capturing the invalidated processors and increment the communication counters that correspond to the invalidated set. The communication counters are reset at the beginning of each sync-epoch. Effectively, as the execution progresses within the sync-epoch, the counters would reflect the processor's communication spatial behavior up to the current execution point within the sync-epoch. At the end of the sync-epoch, the hot communication set is extracted from the counters and stored as a communication signature (bit vector) in a history table called *SP-table*.

When the sync-epoch is a critical section, the communication signature encodes only the ID of the processor that releases the lock. This allows other critical sections that are protected by the same lock to retrieve and use this information as their possible communication target. Note that for noisy instances (Section 3.3.4), no communication signature is stored. Table 7 summarizes how the communication signatures are constructed during the execution.

EVENT	ACTION
Sync-point captured	Retrieve d signature(s) from SP-table Obtain predictor: - If $d = 0$: extract current hot set (after warmup) - If $d = 1$: last hot set - If $d = 2$: last stable hot set - If $d >= 2$: test for pattern (if supported) - If sync-point is a lock : last d processors holding the lock Forward predictor to the L2 controller
RD/WR-miss	- Invoke a prediction action using the obtained predictor.
Confidence alert	- Extract new hot communication set - Replace predictor with new hot set

Table 8: Obtaining prediction.

3.4.3 SP-Table

The SP-table is conceptually similar with the epoch table described in Section 2.5.2. SP-table is an associative table where each entry records a single, per processor, static sync-epoch. Entries are indexed/tagged with the static ID of the sync-epoch and the processor ID. For locks, entries are tagged with the lock variable and are shared by all processors. This allows all critical sections protected by the same lock (in the same or different threads) to share the same communication history.

Each SP-table entry keeps a sequence of communication signatures. This sequence has a bounded size d, the *history depth*. Whenever a sync-point is encountered, SP-table is probed to store the signature of the ending sync-epoch and retrieve the signature(s) of the next sync-epoch. Updates involve shifting out the oldest signature and shifting in the newest. For critical sections, updates occur just after the lock is acquired. This ensures atomic updates in the shared entries and avoids lookups of the table when a processor spins on a lock.

3.4.4 Obtaining Predictions

When a new instance of a previously seen epoch is detected, the associated communication signature(s) are retrieved from SP-table to generate a destination predictor for the misses that will occur

in the new instance. The obtained predictor for the sync-epoch will be forwarded to the processor's L2 cache controller and will trigger an action to the coherence protocol on each miss. The state of the predictor would be simply the previous communication signature or some combination of previous signatures. A summary of how the predictor is formed is given in Table 8. More specifically:

No history available ($d = 0$)**.** If the sync-epoch is met for the first time (or if no history table exists), then history information is not available. In this case, the predictor uses a hot communication set that is extracted from the communication counters while the sync-epoch runs, after allowing some warm-up time, e.g., 30 misses. This would essentially form a predictor that predicts requests based on the activity recorded in the early stages of the interval.

Last hot communication set ($d = 1$)**.** If only one history signature is available so far (or if the table has history depth of one), then the predictor uses the last—and only available—communication signature stored in the corresponding predictor entry.

Last stable hot communication set ($d = 2$)**.** The intersection between communication bit vectors (bit-wise AND) returns the set of destinations that remain stable across the instances. Our predictor combines only the two most recent bit vectors, since this successfully catches stable destinations across consecutive instances, as well as adapts faster to changing stable patterns such as the one shown in Figure 19(b).

Pattern-based hot communication set ($d \geq 2$)**.** A longer history of signatures available to a sync-epoch could further capture hot communication set patterns such as the repetitive pattern shown in Figure 19(c). Specifically, history depth should be at least as large as the repetition distance (or stride) of the pattern, e.g., $d \geq 3$ for the same example. Hardware could detect a repetitive pattern by comparing a new bit vector with all the stored bit vectors, saving the depth s of the one that matches, and correctly predicting the next bit vectors using the one at depth $s - 1$. Our current predictor is tuned to detect only repetitive patterns of stride-2, as it uses a history depth of no more than two.

Lock sync-point. If the captured sync-point is a `lock`, then the retrieved signatures will indicate the sequence of processors holding the lock last. A union of the available d signatures will therefore form a prediction set that includes the last d processors that have held the lock. The predictor may

be further extended to return a union that also includes the bit vector of the preceding sync-epoch, as coarse critical sections are likely to benefit from it.

In order to detect and recover from pathological cases where the predicted communication set does not provide correct prediction, we employ a mechanism that senses low prediction confidence and adapts to a new hot communication set. A recovery step is usually needed in coarse sync-epochs, where communication's spatial behavior could oscillate within a sync-epoch instance. In our current design, the confidence mechanism is a simple 4-bit saturating counter, which increments on correct predictions and decrements otherwise. On each new interval, the counter starts with a high confidence towards the predicted communication signature (counter is fully set) and triggers a recovery step if the confidence level drops below a threshold (counter is zero). To recover, we reconstruct the predictor by extracting the hot communication set of the currently running interval, as it appears up to the current point. The hot set is extracted based on the information recorded in the communication counters that dynamically track the communication activity of the interval.

3.4.5 Integration into the Coherence Protocol

SP-prediction requires additional functionality in the coherence protocol. However, it does not interfere with the base protocol and operates on top of it. We briefly describe how our protocol arbitrates prediction actions, verifies results, and recovers from mispredictions.[2] As a baseline protocol, we use a directory-based MESIF coherence protocol, an extended version of MESI that effectively supports cache-to-cache transfers of clean data [55]. Note that the prediction engine can be integrated into any directory-based protocol, or any snoop-based protocols that can recover from mispredictions [15, 88].

• Requesting node: When an L2 miss for a memory line occurs, a prediction request is generated. The request is sent to the node(s) predicted to have the valid copy of the line and includes a bit identifying it as predicted. The request is also sent to the directory along with a bit vector identifying the predicted nodes.

• Directory: The directory node will receive the bit vector of predicted nodes for every miss and detect whether the targeted set was sufficient or not. Upon detecting a misprediction, it will satisfy

[2]More details on how the protocol handles race conditions and conflicts can be found in similar extensions [15, 109, 2, 3].

the request as it would normally do, resulting in a miss latency similar to the baseline protocol. If the request was for upgrade or write miss with multiple sharers, the directory will invalidate the sharers that were not predicted (if any), and reply to the requesting node, indicating whether the predicted set was sufficient or not and which sharers were correctly predicted.

• Predicted node: When a predicted request for a memory line arrives at the cache controller, the line is searched in the L2 cache. If the line is in Exclusive, Modified, or Forwarding state [55], then a copy of the line is immediately forwarded to the requesting processor. Also, an update message is sent to the directory indicating the new sharing state of the cache line. If the line must be invalidated (i.e., due to request for exclusive ownership), an Ack message is sent back to the requesting processor after invalidation. Otherwise, the cache replies with a Nack message.

• The requesting node will receive responses from the predicted nodes, and also from the directory in case the request was for exclusive ownership (write or upgrade miss). Upon receiving data, the controller will perform line replacement as usual and, if the request was a read, the miss will be completed. If the request was for exclusive ownership, then it will be completed only after the response from the directory and the necessary invalidation Acks from the correctly predicted sharers have arrived (if any). Given that the directory is always aware of the prediction result and can proceed as normal on mispredictions, it is unnecessary for the requesting node to reissue requests.

3.4.6 Discussion of SP-Table Implementation

SP-table can be implemented either in system software or hardware. In the former case, the table is statically allocated at boot time by the OS and kept at a certain memory location. Every sync-point will invoke a trap to the OS, which will handle all necessary operations on SP-table and return a predictor for the upcoming sync-epoch. In a hardware embodiment, a slice of SP-table can be integrated with the L2 cache controller on each processor and hold the information specific to that processor. Table entries that are shared by all processors (for lock sync-points) could be either located at a centralized location on chip, or distributed across the slices in an address-interleaved fashion. All implementations assume that the sync-point's PC, lock address and the processor ID can be extracted at the processor, and the necessary information can be piggybacked and transferred

between the hardware and software components involved.

SP-table has fairly low space requirements. Each slice requires as many entries as the number of static sync-points in an application, which is generally small ($\leq$ 30 + 2$\sim$3 entries as shared portion). Each entry may hold more than one signature, depending on the history depth (we allow no more than two in our evaluation). The length of the signature (in bits) is equal to the number of processors (e.g., 16-bits for a 16-core CMP). Each entry also has a 32- or 64-bit tag (PC) depending on the machine's architecture and an additional bit indicating whether the entry is shared, i.e., a lock. Although each SP-table slice is considered to work as fully-associative, a smaller set-associativity array is also possible without much cost from set conflicts. A 2 KB aggregate SP-table is adequate to hold all necessary information for even the most demanding applications (including 32-bit tags). As we will discuss later in Section 3.5, this size is significantly smaller compared to address- or instruction-based tables.

The location and management of SP-table is an implementation choice that has no significant performance implications, since it is small and accessed relatively infrequently (only on sync-points). A hardware implementation would generally be more appropriate if sync-epochs are short, e.g., if the application has very fine-grain locking. In general, the SP-table design should be dictated by both the design goals and the target application domain.

3.5 EVALUATION

3.5.1 Methodology

To evaluate the performance of the proposed predictor, we extend the system described in Section 3.3.2 with detailed timing models for cache hierarchy and interconnect. The target system is a 16-core tiled CMP with a 4×4 2D mesh network-on-chip (NoC), similar to the model described in Section 2.6.2. Here, each L2 cache bank is private to the processor and inclusive, and coherence is maintacned through a distributed directory-based MESIF coherence protocol with some extensions as described in Section 3.4.5. The NoC operates always at the processor core frequency. Table 9 summarizes the architecture configuration parameters.

For the SP-table, we consider a distributed hardware implementation and each entry can hold

Parameter	*Value*	*Parameter*	*Value*
Proc. model	in-order	L1 I/D Cache	
Issue width	2	Line size	64 B
L2 Cache (private)		Size/Assoc.	16 KB, 1-way
Line size	64 B	Load-to-Use lat.	2 cycles
Size/Assoc.	1 MB, 8-way	Network-on-Chip	
Tag latency	2 cycles	Topology	4×4 2D mesh
Data latency	6 cycles	Router	2-stage pipeline
Repl. policy	LRU	Main mem. lat.	150 cycles

Table 9: Simulated machine architecture configuration.

no more than two signatures ($d = 2$). The SP-table is accessed only on sync-points and the access latency is rarely in the critical path. Updates on communication counters complete in a single cycle, and we account four cycles for extracting a hot communication set. We present the performance of the SP-predictor with respect to the baseline directory protocol and a broadcast protocol. Results consider both serial and parallel sections, although the predictor is effective only during parallel sections. To fairly evaluate a broadcast snoop-based protocol, we assume a totally ordered interconnect with the same configuration as the one with directory. At the end, we compare our prediction approach against a simple locality-based predictor and state-of-the-art address- and instruction-based destination set predictors [88].

Prediction is correct when the predicted set is sufficient to satisfy a communicating miss, i.e., a superset of the sharing information in the directory. The *size* of the predicted set, which is the size of the hot communication set in our case, creates a trade-off between prediction accuracy and bandwidth waste. The fewer the cores included in the predicted set, the less the probability to communicate with the correct core(s) for each request. On the other hand, the more cores in the predicted set, the more redundant messages will be sent, and hence the more bandwidth will be added on the interconnect. In our evaluated scheme, the size of the hot communication set depends on the communication locality of each sync-epoch as explained in Section 3.3.3, and adapts to the changing communication patterns as described in Section 3.4.4.

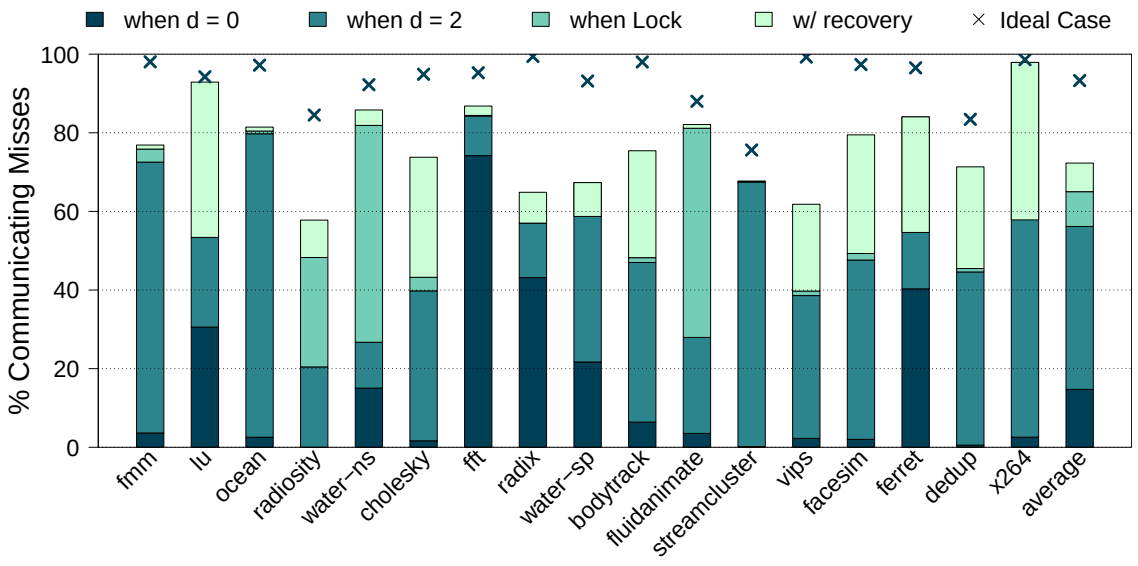

Figure 20: *SP-prediction accuracy:* Percentage of communicating misses that avoid indirection to the directory.

3.5.2 Prediction Effectiveness

Figure 20 shows the percentage of communicating requests predicted correctly. On average, the SP-predictor correctly predicts and eliminates indirection to the directory for 77% of all communicating requests, with 98% (x264) and 59% (radiosity) as the best and the worst case, respectively. The crosses indicate the accuracy that the SP-predictor could obtain ideally, if the hot communication set for each sync-epoch was known a priori. The gap between the actual and the ideal accuracy comes from the lack of predictability in some sync-epoch instances and the sensitivity level of the recovery mechanism. This gap may be bridged somewhat if off-line profiling offers initial prediction information and the sensitivity level is adjusted dynamically.

The percentage breakdown indicates the prediction accuracy when different information was available to the SP-predictor. The bottom stack accounts for correct predictions made when no information from past sync-epoch instances was available. Such situations appear in applications where major sync-epochs are not replayed (fft, radix, and ferret). In these cases, the predictor relies mostly on most recent within-interval communication activity to predict miss targets. The next two stacks correspond to misses correctly predicted based on signatures from past sync-epochs,

Benchmark	Avg. Actual Targets per Req.	Avg. Predicted Targets per Req.	Ratio of Predicted to Actual
fmm	1.19	3.11	2.61
lu	1.01	2.46	2.46
ocean	1.08	3.15	2.94
radiosity	1.11	4.12	3.71
water-ns	1.41	2.53	1.80
cholesky	1.04	1.89	1.83
fft	1.01	2.37	2.36
radix	1.00	2.75	2.75
water-sp	1.58	2.75	1.75
bodytrack	1.13	2.8	2.49
fluidanimate	1.14	2.05	1.79
streamcluster	1.14	1.95	1.72
vips	1.01	2.06	2.05
facesim	1.04	2.56	2.47
ferret	1.01	1.14	1.13
dedup	1.10	2.34	2.15
x264	1.01	1.93	1.93

Table 10: Average actual and predicted set size.

indicating separately those occurring within critical sections. Applications with highly repeatable sync-epochs such as ocean and streamcluster can take advantage of the pattern-based prediction policy. Similarly, applications with fine-locking such as water-ns and fluidanimate gain with highly accurate predictions due to the ability of our predictor to retrieve the random sequence in which threads execute the critical sections. On average, those sync-epoch history-based predictions account for up to 40% in prediction accuracy. Sync-epochs with unpredictable intervals will eventually adapt their predictors based on the recovery mechanism and correctly predict an additional 9% of requests on average.

Messages will be wasted if the predicted target set for a miss is incorrect, or larger than the minimum sufficient target set. Table 10 summarizes the differences between the minimum and the predicted average target set size. The minimum sufficient set size is generally close to 1 since read requests, which are the majority, must always contact only a single destination.[3] By comparing separately the reads and writes, we found that, on average, the predicted set includes 1.4 and 0.5 more targets per request respectively. More insight on how the prediction affects the bandwidth

[3]The reported numbers assume a cache-to-cache transfer request for clean data to have a sufficient set size of 1, which is not necessarily true in a MESIF protocol [55].

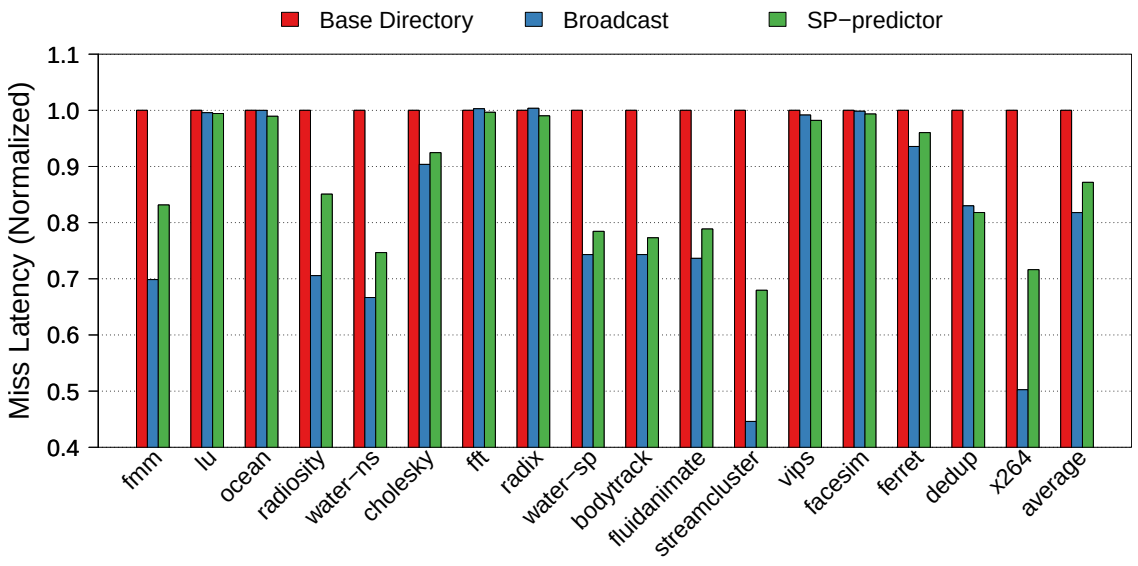

Figure 21: Average miss latency. (Note: Y-axis starts at 0.4.)

demands is given by more detailed results presented in the next subsection.

The way the hot communication set is extracted (Section 3.3.3) strongly affects the trade-off between latency and bandwidth. The current policy leads to some bias towards higher bandwidth when the locality is poor, since there are no strict bounds on the maximum size of the set. In general, the policy can be tuned depending on the design goals and requirements. For example, in a case where bandwidth demands must be bounded to avoid exceeding a power envelope, one could tune the policy to extract a hot set that does not exceed a certain size.

3.5.3 Performance Results

Impact on miss latency. Correct predictions will satisfy misses without paying the cost of indirection to the directory, thereby reducing the average cache miss latency. Incorrect predictions are detected by the directory, which will then satisfy the miss without noticeably degrading the latency of the indirected miss. Figure 21 shows the average miss latency achieved by the SP-predictor and the baseline protocols. Average latency is calculated by treating each miss individually, and results are normalized to the directory protocol. The results show that on average, SP-prediction reduces miss latency by 13% relative to the directory protocol and attains up to 75% of what the broadcast

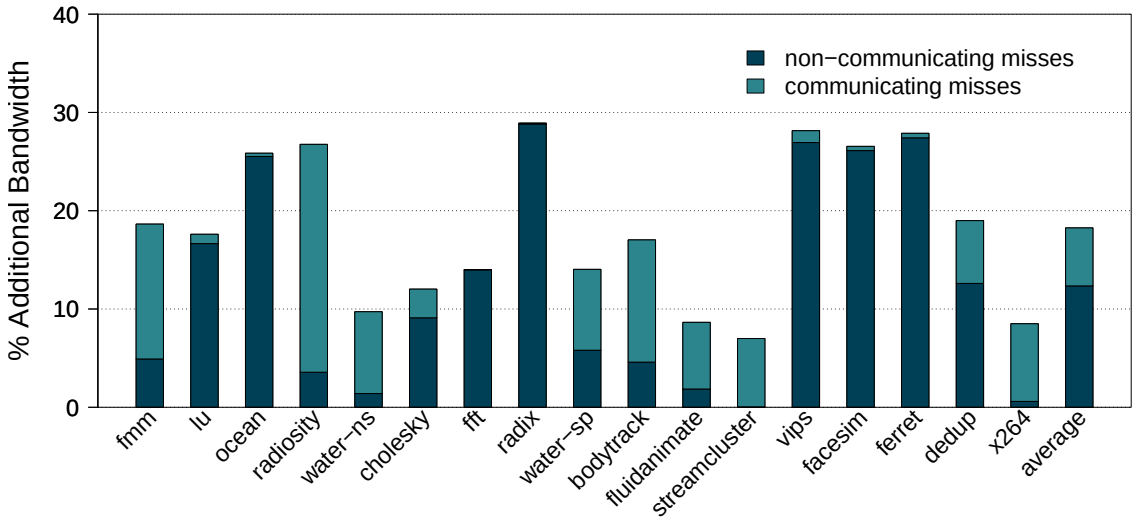

Figure 22: Additional bandwidth demands of SP-prediction relative to the base directory protocol.

snooping protocol can achieve. Under the (true) assumption that the NoC does not get severely congested, the broadcast scheme approximates the ideal case in terms of miss latency.

The predictor predicts correctly and reduces the latency for both read and write requests. A correctly predicted "read" has slightly higher impact compared to a correctly predicted "write", as writes may have multiple targets to reach and wait for acknowledgments. Also, the prediction accuracy slightly declines as the number of targets increases. Nevertheless, write requests with multiple targets are generally a small fraction of the overall misses, and their impact on the overall reductions in latency is limited.

Marginal improvements in some applications (e.g., lu and radix) are due to the limited fraction of communicating misses (recall Figure 14). The smaller this fraction is, the fewer the opportunities for latency reduction. Moreover, the high miss latency of non-communicating misses (i.e., off-chip misses) will, in the end, overshadow the improvements coming from accelerating on-chip, communicating misses. A quick look at how this fraction varies across the applications clearly explains why the miss latency reduction is limited for each application. Note that this also limits the effectiveness of the broadcasting scheme. It is generally possible for a larger cache size to elevate the fraction of communicating misses for memory bound applications, and hence increase

the impact of the predictor to the miss latency reduction. Sensitivity analysis of cache parameters and workload input sizes (not reported in this work) have shown expected observations and trends.

Impact on bandwidth requirements. To measure the impact of target prediction on bandwidth, we track the number of bytes transmitted on the NoC due to L2 cache misses. These include request messages to predicted cores, request and update messages to the directory, and control and data responses. Figure 22 shows the additional average bandwidth requirements of a coherence request, relative to those of the baseline directory protocol. The results show that SP-prediction increases the bandwidth requirements by 18% compared to the baseline. The snooping protocol would have the highest bandwidth demands since messages are broadcast to all targets on each miss, whereas the directory protocol essentially approximates the ideal case possible. Overall, SP-prediction keeps its additional bandwidth requirements below 10% of what the broadcasting protocol would additionally demand from the baseline directory protocol (the actual bars for broadcasting are not shown due to the very large difference).

Much of the additional bandwidth comes from the (always unfortunate) attempts to predict non-communicating misses. This portion is shown by the bottom stack and accounts for 70% of the overhead. Applications with a large fraction of non-communicating misses will therefore increase the bandwidth demands with no positive return in latency. Prior work has shown that most of such attempts can be detected and avoided by simple snoop filtering [91]. For example, a simple low cost TLB-based snoop filter can detect $\sim$75% of them [33]. Thus, the use of orthogonal techniques can substantially reduce the associated bandwidth overheads without compromising the latency improvements.

Impact on execution time. Figure 23 depicts the overall improvements in execution time as a result of reducing miss latency. SP-prediction improves the execution time by 7% on average, with x264 seeing the best improvement (14%). Depending on the interconnect design and control parameters, an excessive traffic could congest the network and affect the performance negatively. In our simulated system, congestion levels remain low for both the prediction-augmented directory protocol and the base broadcast protocol. Marginal negative impact was observed for broadcasting only in applications with a very small fraction of communicating misses.

Impact on energy. We estimate the energy impact of SP-prediction using an intuitive analytical model that considers the dynamic energy consumed on the interconnect and L2 cache snoops. For

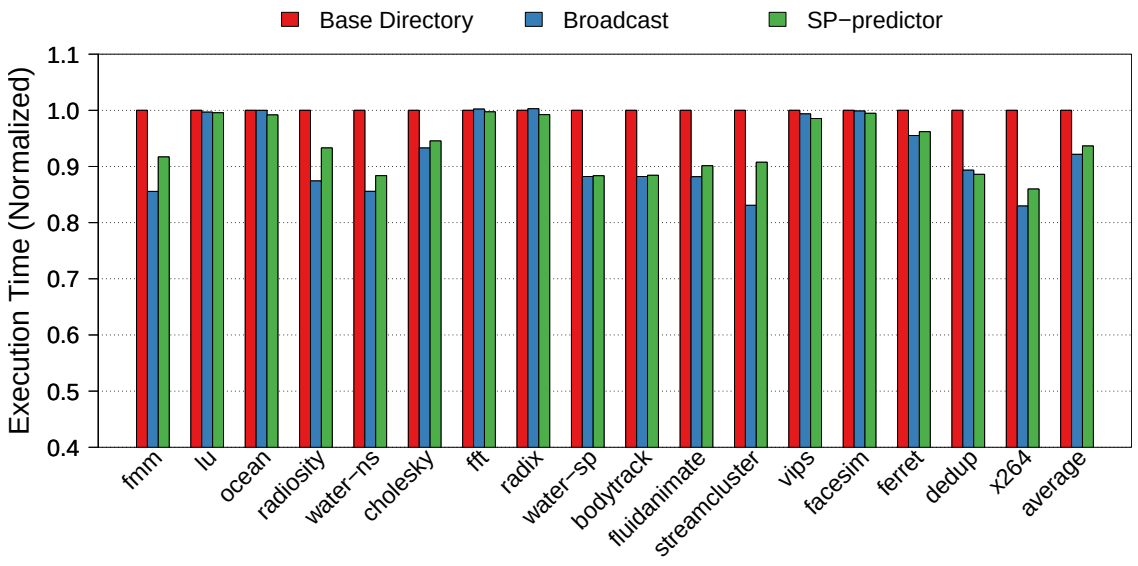

Figure 23: Execution time. (Note: Y-axis starts at 0.4.)

the network, we assume that the energy consumed is proportional to the amount of data transferred [9]. We also assume that the energy consumed in a router is four times that consumed in the link. For cache snoops, a single cache tag lookup energy is estimated using CACTI [48], assuming a 32nm technology. Figure 24 presents the normalized results. Enabling SP-prediction over a directory protocol increases the energy requirements on network and cache lookups by 25% in total. Yet, this is substantially lower than the energy requirements of a snoop broadcasting (2.4×). Considering that a large fraction of traffic and snoop overhead could be filtered, as discussed previously, the new energy demands could be brought down to below 8%.

3.5.4 Comparison with other Predictors

We compare SP-prediction with address- and instruction-based prediction, implemented according to the "group" destination set prediction model proposed by Martin et al. [88]. In addition, we compare it with a simple locality-based predictor that uses no index, i.e., predicts simply based on the coherence activity of previous misses, independent of their address or instruction. For abbreviation we will refer to them as ADDR-, INST-, and UNI-prediction, respectively. The ADDR and

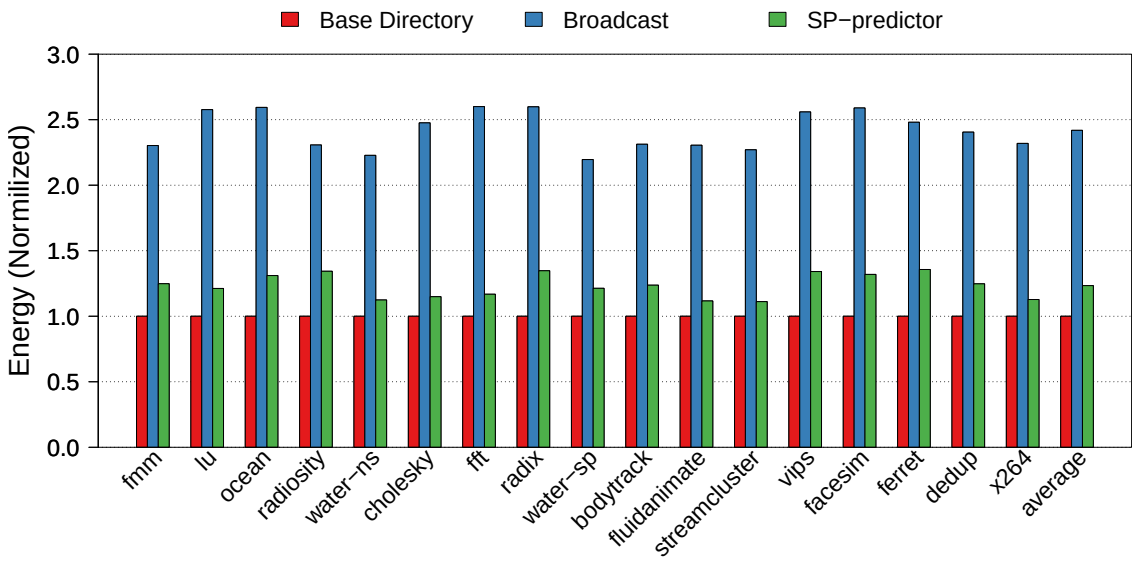

Figure 24: Energy consumed on NoC and cache lookups.

INST prediction models use both external coherence requests and coherence responses to train a predictor for each data block or instruction. The UNI predictor uses only the coherence responses, i.e., it is trained based on the targets of previous misses by the same core.

All the predictors return a group of possible sharers, aiming at high prediction accuracy while making best efforts to keep the bandwidth requirements small[4]. Each predictor entry incorporates a two-bit counter per core that accumulates the recent activity towards each destination, and a train-down mechanism that ensures that the predictor eventually removes inactive destinations [88]. For a 16-core machine, each group predictor entry requires a total of 37 bits (tag not included): 32 bits for the train-up counters and a 5-bit roll-over counter for the train-down purposes. For SP-prediction, we consider an SP-table with two signatures per entry (total of 33 bits) as a fair setting for comparison. Note that SP-prediction also requires a set of communication counters (1-byte each) and a predictor register, which account for a fixed cost of 17 bytes per core.

Each predictor represents a point in the trade-off between latency and bandwidth. To effectively visualize this trade-off, we plot results on a two dimensional plane (Figures 25, 26). The

[4]Other prediction policies such as “owner” or “group/owner” can also be used and fairly compared as far as all predictors are tuned to the same base policy.

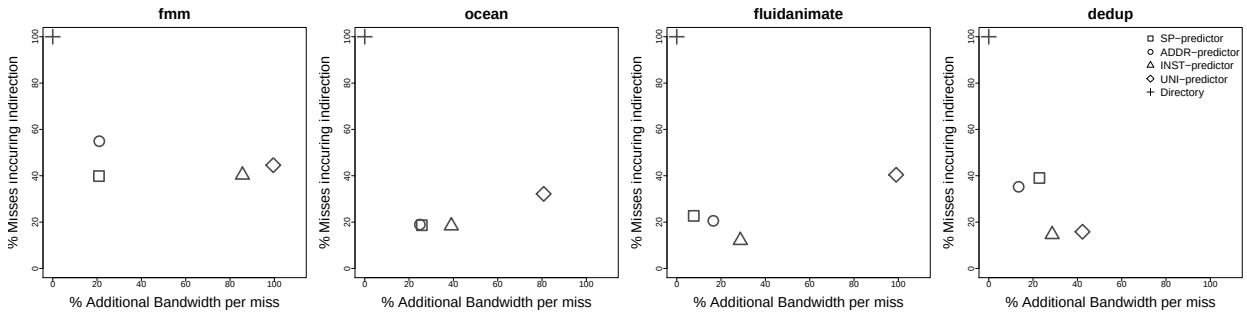

Figure 25: *Performance/bandwidth trade-off comparison:* The lower-left corner represents the best point on the trade-off space. The results are expressed relative to the directory-based protocol, which is indicated with a "cross" symbol in the upper-left corner.

horizontal dimension represents *request bandwidth per miss* (in addition to that of the based directory). The vertical dimension represents *latency*, measured as the *percent of misses that require indirection*. The chosen metrics provide a desirable level of detail for deriving insightful results for the performance of the predictors under consideration.

Figure 25 displays the results for the four predictors in four different applications for illustration. The results assume predictors with an infinite number of table entries for their indexed tables, i.e., they do not consider space efficiency. Overall, SP-prediction lays in the trade-off plane comparably to address- and instruction-based prediction. Among the examples, fmm presents a case in which SP-prediction outperforms all other predictors, achieving both higher accuracy and lower bandwidth. In contrast, dedup presents a counter case, where SP-prediction is weaker along the accuracy dimension. Accuracy levels between ADDR and INST appear to be similar, with the ADDR-predictor having more tendency towards lower bandwidth requirements. UNI-prediction is shown to have lower accuracy, which also negatively affects the bandwidth demands since incorrect predictions place unnecessary messages on the interconnect.

Each scheme has, however, very different space demands to meet the illustrated maximum performance. A perfect ADDR-prediction scheme suggests storage requirements in proportion to the size of the memory blocks, which is prohibitively large. Common practice is for ADDR to consider, instead, predictors per macro block (e.g., 256-bytes in our implementation). This reduces the maximum space requirements, and further improves the predictor by capturing spatial

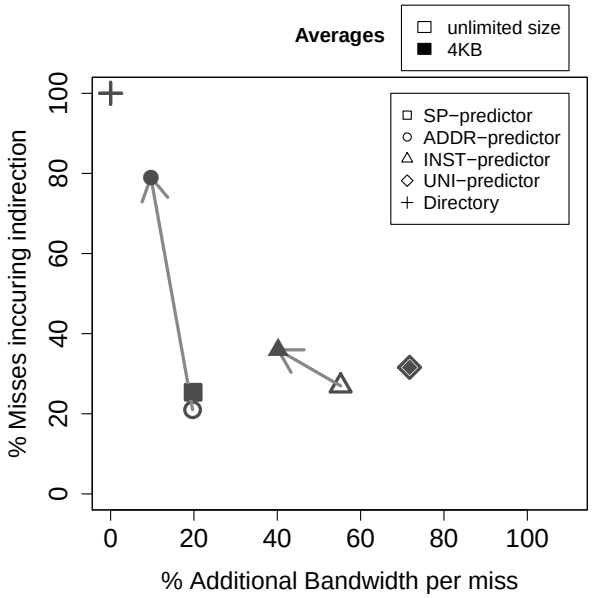

Figure 26: *The effect of space requirements to prediction performance:* SP-prediction and UNI-prediction are not affected since they have significantly lower space requirements.

locality. However, even with macro-blocks, the number of entries required to achieve the maximum performance is in the order of Kilo. INST has been promoted for its low storage needs; however, it requires significantly more table entries than the SP-table (equal to static load/stores). UNI-prediction requires only a single prediction entry and represents the cheapest possible solution. The SP-prediction's storage requirements are inherently bounded by the number of static sync-points of the application as shown in Table 6. This corresponds to substantially lower space demands compared to ADDR and INST. Assuming that the SP-table is easily implementable in the software layer, its hardware space requirements can be largely eliminated, reaching those of UNI-predictor.

To evaluate the sensitivity of the predictors to space requirements, we implement them with limited number of table entries. Figure 26 compares the performance of different predictors when table entries go from unlimited to a finite number of 512 ($\sim$4KB of storage space). To simplify the illustration, we only show the average results for each predictor, over all the studied applications. The results indicate that limited space yields lower accuracy for ADDR and INSTR compared to SP-prediction. Nonetheless, they present a corresponding decrease in bandwidth, since prediction is attempted on fewer misses.

The prediction performance per space requirements is in a sense the measure of how well the prediction information is encoded, or in other words, the measure of a predictor's space efficiency and cost. Considering that SP-prediction requires significantly smaller storage than ADDR and INST, we argue that the reported small performance differences are insignificant when space and power requirements are a primary design constraint, as is the clear case in modern and emerging CMP implementations [35]. In conclusion, from the space requirements perspective, an SP-predictor with ∼256 entries can achieve performance equivalent to INST with ∼1K entries, or macro-block ADDR with ∼8K entries, on average.

3.5.5 Discussion

Predictor's power consumption comparison. Prediction tables consume static and dynamic power. Static power is proportional to the table size, which is substantially smaller with SP-prediction. Dynamic power is primarily affected by the associativity, and the access frequency of the predictor tables. While the ADDR and INST access their tables on every miss, SP-predictor keeps the prediction set in a single register, and accesses the SP-table for updates only on sync-points. This directly translates into power savings. Based on an overall observation, the SP-table would be accessed once for every ∼300 accesses of an ADDR- or INST-based table.

Thread migration. So far we have assumed that communication signatures and predictors consist of bit vectors representing target physical cores. If thread movements are allowed between cores, then those representations should track a "logical core-ID" (e.g., thread-id) rather than physical ID. The logical-to-physical destination mapping must be known at the core side, and could be applied before or after the formation of the predictor, depending on the coherence controller implementation.

Projections for commercial workloads. Database, server, and OS workloads are mostly based on lock synchronization and as a result have less regular and predictable communication patterns [108]. The proposed SP-predictor can effectively predict the communication activity within critical sections since it can retrieve communication signatures on lock points that include the cores (or the sequence of cores) holding the lock previously in time. Results from applications with a high count of critical sections (e.g., fluidanimate and water-ns) show high prediction accu-

racy for the misses occurring within critical sections (Figure 20). Therefore, although we have not performed experiments on such workloads, we expect our predictor to work reasonably well.

3.6 RELATED WORK

Address and instruction-based indexing have been the basis of hardware coherence predictors [92, 70, 61, 62]. In the context of destination set prediction, Acacio et al. [2] studied a two-level owner predictor where the first level decides whether to predict an owner and the second level decides which node might be the owner. In a similar work, they study a single-level design to predict sharers for an upgrade request [3]. Bilir et al. [15] studied multicast snooping using a "Sticky Spatial" predictor. Martin et al. [88] explored different policies for destination set predictors to improve the latency/bandwidth trade-off under ordered interconnects. Other studies have further explored the impact of predictor caches [93] and perceptron-based predictors [79].

There have been numerous other efforts to improve coherence performance. Many protocols were developed or extended to optimize for specific sharing patters, such as pairwise sharing [54], migratory sharing [23, 111], producer-consumer sharing [20] and some mix of those [43]. Dynamic self-invalidation was proposed to eliminate the invalidation overhead [75, 70]. Alternatively, software-driven approaches have proposed programming models or utilized compilers to effectively *prefetch* or *forward* shared data to reduce miss latencies [67, 116, 1]. A thorough characterization of data sharing patterns and inter-processor communication behavior in emerging workloads is presented in a work by Barrow et al. [10].

More recent work has exploited properties relevant to CMP architectures to accelerate coherence, such as core proximity and fast and flexible on-chip interconnect. Brown et al. [16] describe an extension to the directory-based coherence protocol where requests are first sent to neighboring cores. Barrow et al. [11] propose adding new dedicated links for forwarding the requests to the nearby caches, delegating directory functions in case of proximity hits. Various other proposals, such as Token Coherence [89], examine novel approaches on maintaining coherence in unordered interconnects without requiring directory indirection. Eisley et al. [32] propose to embed directories within the network routers that manage and steer requests towards nearby sharers. Jerger et

al. [34] propose a virtual tree structure to maintain coherence in an unordered interconnect, with the root of the tree acting as an ordering point for requests. In Circuit-Switch Coherence [59], the same authors show how coherence predictors can leverage existing circuits to optimize pair-wise sharing between cores. Similar to virtual tree coherence, DiCo-CMP [98] delegates directory responsibilities to the owner caches.

3.7 SUMMARY

Predicting target processors that a coherence request must be delivered to can improve the miss handling latency in cache-coherent shared-memory systems. In this work we study and propose *Synchronization Point based Coherence Prediction* (SP-Prediction), a new run-time scheme for predicting coherence communication targets of private cache misses. SP-prediction employs mechanisms that capture synchronization points at run time, track the communication activity between them, and extract simple communication signatures that guide target prediction for future misses. The new scheme is substantially simpler than existing techniques because it exploits the inherent characteristics of an application to predict communication patterns. Compared with address- and instruction-based predictors, SP prediction requires smaller area and consumes less energy while achieving comparative high accuracy. We anticipate that the synchronization point driven prediction approach could be applicable to further communication optimization cases.

4.0 THREAD CRITICALITY PREDICTION

4.1 INTRODUCTION

At a synchronization point, a processing core may need to wait for one or more cores to complete certain part of execution before it can proceed. This may stall the core for a considerable amount of time, limiting performance and wasting energy. Minimizing the waiting times at synchronization points requires distributing the workload equally between the available cores, and limiting the impact from contention on shared resources and other machine-dependent delays.

Load balancing is traditionally a responsibility of the software [24]. After decomposing the work into parallel tasks, the programmer, compiler, or runtime must distribute those tasks to threads in order to achieve optimal performance. Furthermore, because task scheduling impacts other aspects of the program execution such as data locality, scheduling should target for the most efficient usage of resources.

Although load balancing is conceptually easy, it is practically hard to optimize due to the several trade-offs related to the task granularity and scheduling. In addition, synchronization and communication overheads, and other architectural effects during execution create random delays that directly affect the execution of tasks, making the balancing process inherently limited. Even approaches that perform load balancing dynamically have difficulties in achieving a good balance (e.g., where to steal work from) while they may introduce additional overheads.

Identifying the resulting execution progress imbalances among synchronized threads in a parallel program has important ramifications for performance and resource management. If for example a thread that appears to run slower than the rest of the threads (a.k.a. critical thread) is identified, then its relative difference can be amortized by re-considering the software design choices (e.g., [29]), or by dynamically adapting the hardware itself accordingly [80, 83, 17, 71, 12, 29].

Examples include temporarily boosting the frequency of the core where the critical thread is running on [6], migrating it to a faster core [112, 60], raising the fetch priority of that thread in an SMT context [17], or by allowing more task stealing from that threads in a task stealing context [12]. Inversely, thread criticality can be amortized by slowing down non-critical threads to save for example power without affecting the performance.

In this work we characterize thread criticality in various barrier-based workloads, and we identify the underlying reasons that cause the threads to reach barriers at different times. We show that although tracking a particular metric is, in certain cases, enough to predict the criticality at run time, it is not robust across different workloads and architectures. To address this limitation, we present a new thread criticality predictor that improves prediction accuracy by using additional, history-based information associated with each global synch-epoch. By utilizing the repeatability of global synchronization points, the new scheme is able to predict critical threads at run time on their natural granularity and with higher confidence. Using the new criticality predictor, we are able to improve performance or significantly mitigate wasted resource usage by dynamically scaling the frequency/voltage of the on-chip cores.

4.2 BACKGROUND AND RELATED WORK

4.2.1 Forms of Parallelism

Thread criticality is, in part, related to load balancing. Therefore, it is important to briefly review the forms of parallelism and load distribution. There are two main general forms in which computation is explored for parallelism: Data parallelism and function parallelism [24].

Data parallelism refers to exposing concurrency among data. For example, a loop whose iterations can be executed in parallel is a form of data parallelism (in this particular case also called loop-level parallelism). Parallelizing loops often leads to similar (not necessarily identical) operation sequences being performed on a large data structure. Loop-level parallelism accounts for a large fraction of kernels in parallel programs and is usually easy to expose as it is a more structured form of parallelism. Many parallel languages natively support loop-level parallelism.

Function parallelism[1] refers to situations where entirely different calculations can be performed concurrently on either the same or different data. Generally, function parallelism covers a more dynamic form of parallelism where a parallel task can be spawned at any point. For example, analyzing dependencies on a tree-based structure where parent-nodes depend on the results of children-nodes can generate independent computations that can be performed in parallel. Pipelining is another form of function parallelism in which the different pipelined operations are performed concurrently.

Data parallelism and function parallelism are often available together in an application. The degree of data parallelism usually grows with data set size, while function parallelism is usually modest in data parallel applications. Function parallelism is also usually more difficult to exploit in a load-balanced way, since different functions involve different amounts of work and have different scaling characteristics. Most parallel programs that run on large-scale machines are data parallel and exploit function parallelism mainly to reduce the amount of global synchronization required between data parallel computations.

4.2.2 Task Assignment for Load Balance

To exploit concurrency, tasks should be distributed among threads in a balanced way. Determining the best assignment for optimal performance is a non-trivial task as the latency of tasks is often based on the problem size and is highly platform or hardware dependent. There are many decisions to be made when mapping a parallel program to a platform. These include determining how much of the potential parallelism should be exploited, the number of processors to use, how scheduling should be performed, etc. The right choice depends on the relative costs of communication, computation, and other hardware costs, and varies from one multicore platform to another.

This mapping can be performed manually by the programmer or automatically by the compiler or run-time system. Given that the number and type of cores is likely to change from generation to the next, finding the right mapping for an application is hard and far from general. The literature distinguishes between static and dynamic task scheduling approaches based on whether the assignment is predetermined or can change at run time [24].

[1] Also called task parallelism, though this is an overloaded term.

A static assignment is typically an algorithmic mapping of tasks to processes which once determined, doesn't change again at run time. To achieve a good load balance, a static task assignment requires that the relative amounts of work in different tasks be adequately predictable, while ensuring that other environmental conditions do not perturb the relationships among threads (e.g., interference from other applications). Static task scheduling and load balancing usually require a skillful programmer (or compiler) to determine the optimal task assignment based on a detailed analysis of the target architecture and an off-line application profiling. This analysis is complex and its complexity increases with the number and the type of processor hardware contexts, the depth of resource sharing, and the number of simultaneously running tasks. In addition to this, any change in the application or in the hardware platform requires the repetition of the whole analysis.

In dynamic task scheduling, tasks are distributed among executing threads by the underlying library or run-time environment. This enables convenient expression of dynamic tasks in the program and significantly improves the productivity in parallel programming. With dynamic tasking, computation is divided into tasks that are not pre-assigned to threads but are maintained in a pool. At run time, each thread repeatedly takes a task from the pool and executes it (while possibly inserting new tasks into the pool), until no tasks remain.

Dynamic task scheduling generally provides good load balancing despite workload's and machine's unpredictable artifacts. However, it introduces overheads as the scheduler has to manage parallelism during the actual execution (e.g., discover and preserve dependencies among tasks, manage the task pool, implement the control/scheduling policy). These overheads grow as the task granularity shrinks, trading-off the potential for better load balance. Also, dynamic scheduling disables the explicit control of assigning the tasks to specific processors, which could eventually increase communication and compromise data locality.

Static load balancing is usually preferred when it can provide good load balancing as it does not introduce additional overheads for automatic task management [24]. Pthreads is a popular example of a parallel model that supports static assignment and on which many applications from the scientific and real-time domain have been written. On the other hand, dynamic task parallelism is included for mainstream use in many new programming models for multicore processors and shared memory parallelism, such as Cilk [37], OpenMP 3.0 [7], Java Concurrency Utilities [47], Intel Thread Building Blocks [97], and Microsoft Task Parallel Library [77]. Scheduling algo-

rithms based on Cilk's work-stealing scheduler [37, 78] are gaining popularity due to their dynamic lightweight task parallelism.

4.2.3 Related Work

There are numerous works proposing task scheduling approaches for addressing the load balancing problem. In the context of software controlled dynamic scheduling, most recent work focuses on improving task-stealing scheduling techniques, as those techniques yield a good trade-off between balance improvements and management overheads [78, 7, 97].

Hardware techniques can further improve load balancing while providing full transparency, fine-granularity tracking, and low overheads [68, 12]. Some techniques, such as Carbon [68] and [4], provide hardware support for task queuing and stealing. Other techniques provide models that can identify imbalances at run time, and alleviate the imbalance through architectural techniques or hints back to the software [12].

Identifying critical threads is essential for effectively achieving a good execution balance among related threads. Critical thread is defined as the thread that causes other threads to wait the most. Hardware-based approaches that can identify or predict critical threads at run time can be proved important as fine-grain parallelism becomes more common. Proposed thread criticality predictors can detect threads lagging behind and re-balance the available parallelism on-the-fly using techniques such as work-stealing and DVFS, or OS re-scheduling policies [71]. Thread criticality predictors have been correlated with miss rate [12], relative instruction count [71], relative loop count [17], and barrier-based history of execution times [80, 83].

Most thread criticality predictors assume Single-Program-Multiple-Data (SPMD) type of parallelism; therefore, they are not expected to work well under function parallelism. Bhattacharjee et al. [12] propose a criticality predictor that achieves good accuracy by tracking the threads with higher miss rate. However, this predictor works well only under the assumption that no imbalances occur in the absence of miss rates. This does not always hold, especially in applications with function parallelism, in machines with out-of-order cores, in distributed cache organizations, or in systems with heterogeneous features in general.

More recently, Du Bois et al. [29] proposed "stack criticality prediction" that predicts the crit-

ical thread based on the amount of time each thread spends in useful work relative to co-executed threads. This approach does not consider the intermediate barriers but rather focuses more on a coarse-grain balancing of the load. Also, similar to previous approaches, it concentrates on one particular observation and does not take into consideration various other random delays that appear during execution.

4.3 CHARACTERIZING CRITICALITY

4.3.1 Definitions and Optimization Opportunity

We define as *wait-stall time due to execution imbalance* $T_s(t, e)$ the time a given thread t spends "waiting" for one or more threads to reach a common synchronization point, including the end of the program where all threads must finish together. Considering that threads are subject to various common synchronization points (e.g., barriers) throughout their execution, execution imbalances naturally occur at the epoch granularity and must be examined within epoch-intervals. Before a thread is put into a hold, the thread is considered to be doing useful work which we define as *compute time* $T_c(t, e)$. The variation of the compute time $T_c(t, e)$ among the co-running threads during an epoch e is a measure of the execution time imbalance among threads, or *thread criticality*. Figure 27 illustrates the concepts described.

Different threads may experience different wait-stall times during an epoch. Ideally, balanced threads will reach the barrier without significant time difference. However, when this difference is significant, the slowest thread creates a "critical path" that eventually defines the overall performance of the application. The thread that forms the critical path is commonly called a critical thread. In general, the order in which threads reach a common synchronization point expresses the thread criticality order.

During T_s, threads only waste time and resources. Optimizing for criticality refers to reducing T_s. This can be achieved by either slowing down the least critical thread(s) (possibly saving energy resources), or accelerating the most critical thread(s), which are the ones responsible for the delays. Essentially, the sum of $T_s(t, e)$ across all epochs e is the best possible time an application can gain

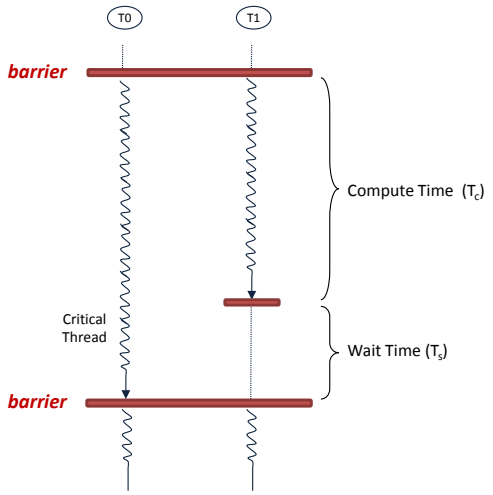

Figure 27: *Notion of compute-time imbalance between co-executing threads (a.k.a. thread criticality).*

in total if the critical threads are accelerated, while the ratio of this sum to the total execution time is a rough indicator of the potential energy that could be saved if the CPU was operating in low power mode during such periods.

Figure 28 shows the gap between the execution time as defined by the most and least critical thread, for the applications studied here. The difference reflects the maximum potential improvement in overall execution time if all threads could reach synchronization points and finish computation at the same time as the least critical thread. In some applications, this difference is quite significant. For example, water.nsq could potentially complete its execution in almost half time in the absence of thread criticality.

4.3.2 Sources Causing Thread Criticality

Understanding the reasons behind thread criticality is essential for developing effective techniques to identify and predict critical threads and potentially guide fine-balancing optimizations at run time. Although software employs systematic load balancing techniques, execution time imbalances among threads still occur at run time causing a considerable amount of wait cycles wasted at synchronization points. From the software point of view, this is due to the inability of a load

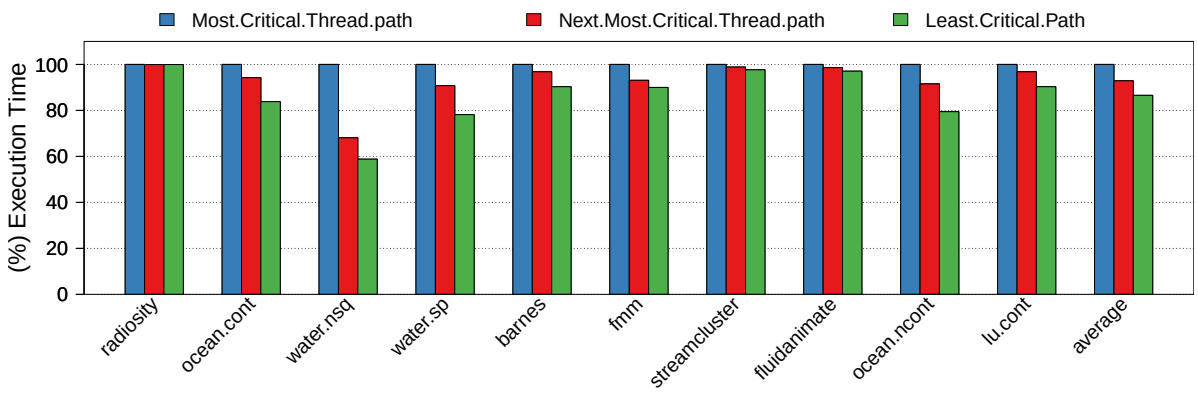

Figure 28: *Overall compute-time imbalance in various applications*: The gap reflects the room for potential performance and/or energy optimizations.

scheduler (both static and dynamic) to track the fine-grain implications that cause criticality. To uncover the real sources of criticality at run time, however, we must investigate further the dynamic program behavior and how it is linked to the creation of such execution time variations. In this section, we study the possible sources that disturb the computation balance across threads, and we present quantitative example cases that demonstrate the relations of program behavior to thread criticality.

Thread criticality at run time occurs due to the following reasons:

- *Unequal work assigned to different threads.*

  Each thread is assigned a certain task or a set of tasks to complete. These tasks may involve a different problem (functional parallelism), or a different data set for the same problem (data parallelism) which cannot be naturally divided into sub-problems of perfectly equal size. Sub-problems with different functionality or different problem size assigned to different threads will naturally lead to load imbalances.

  This type of workload balance is largely a responsibility of the software; however it is usually impossible to guarantee perfect load balance as there are often assumptions regarding the problem sizes and the underlying computing resources. This is especially true for unstructured programs and programs that employ functional parallelism, even when dynamic scheduling is

performed.

- *Unequal data localities (cache performance).*

  The time needed for a memory reference to complete introduces delays in computation that depend both on the size and distribution of the working data set, as well as on the machine's architecture and configuration. A core is likely to experience different cache miss rates relative to other cores if it exercises instruction and data sets of unequal size or of different temporal and spatial locality patterns. Even when the computational load appears to be balanced, such variations in cache performance can cause quite significant performance imbalances. Software balancing techniques usually consider this factor before making decisions. In addition to the data access patterns, the platforms on which multithreaded programs usually run have distributed cache resources, which may cause dramatic variations in hit latencies and proximity to data for each core. Consequently, machine architecture and configuration has a particular impact on thread criticality. Furthermore, the presence of defects and other process related variations on the hardware could further impact the cache performance balance.

- *Random delays.*

  Accesses to shared resources: Threads compete for shared system resources such as caches, chip interconnect and memory controllers. The interference can unpredictably affect aspects that add to the performance imbalance such as access latencies, data localities, and memory level parallelism.

  Inter-thread communication: Although data sharing in shared-memory applications is well understood, the underlying core-to-core or cache-to-cache communication is as dependent on the machine's coherence architecture as well as on the non-deterministic nature of parallel execution.

  Lock contention / serialization: Heavy use of locks or a coarse grain locking may lead to significant contention on entering critical sections, causing threads to serialize execution and waste cycles. This becomes especially prevalent as the number of concurrent threads increases. Previous studies have shown that lock contention can become the dominant bottleneck in scalability for many existing parallel applications [112].

  Because threads request and compete for the locks in a non-deterministic way, it is difficult

to guarantee that the threads will get an equal share in waste cycles when attempting to enter a critical section. Some threads may get more "lucky" and enter the critical sections without facing competition, while in other cases they could face the worst case of serialization. As lock contention becomes more significant in determining thread's performance, becomes also more significant in sourcing thread criticality.

System interference: The operating system can interfere with the running application. For example, it may schedule out a thread that is spinning for a long time, or execute some unpredictable kernel operation, or switch the context to another user-level thread, etc. Certain actions or frequent activity by the OS can cause unpredictable delays to the application.

Interference from other co-running user-level applications: Beyond the possibility of a thread being preempted, other user-level threads may implicitly interfere with the program by competing for the same hardware resources, in the same manner that related threads compete. However, unless some level of quality of service is guaranteed, co-executed programs may have devastating effects in thread criticality.

- *Heterogeneous architecture characteristics.*

  Heterogeneous architectures employ components with different performance characteristics aiming at achieving better performance per watt. Architectures with heterogeneous cores have been proved particularly effective for environments in which many single-threaded applications run concurrently. Performance improvements are also possible for multithreaded applications if the serial sections of the application can be accelerated using a faster core while the parallel sections are distributed to a high count of slower cores to exploit parallelism.

  However, if parallel threads are assigned to run on cores with different performance characteristics, it is obvious that scheduling the parallel load effectively (i.e., achieving performance scaling) becomes much more challenging.

  Imbalances will be immediate if existing, optimized parallel programs run instead on systems with heterogeneous processing elements. An attempt to statically re-optimize such programs according to the underlying heterogeneity (through profiling and simulations), would require a tremendous amount of effort and would yield solutions that are not well suited across different platform configurations and problem input sizes.

Perhaps the only viable option of the software to provide load balancing in heterogeneous environments is to dynamically schedule the parallel tasks to heterogeneous cores (e.g., via task stealing). Despite the fact that dynamic scheduling can potentially close the large gaps in performance imbalance, it has its own limitations as previously explained, and is more complicated to be realized in heterogeneous systems

- Unintentional heterogeneity:

  Apart from heterogeneity by design, some level of heterogeneity is naturally present even in systems with identical cores, as process variation often yields cores with slightly different characteristics (i.e., variations in operational frequency). Although vendors guarantee that such variations are within certain limits, slight variations may have observable effects in compute intensive parts of the code, and possibly exacerbate the imbalance problem.[2]

4.3.3 Epoch Cycle Stack

Compute time, which is the total time required by the thread to complete the scheduled tasks during an epoch, is formed by various different time-impacting components. These include the time consumed for computation, the cache miss delays, and all other random delays as described above. Figures 29– 32 present actual examples of how different sources of thread criticality could affect particular epochs. Each bar represents the cycle stack of a particular core, and is broken down to the amount of cycles spent on 1) cpu-internal computation (including fetch, execute, branch misses, etc.); 2) stalls due to outstanding misses (which include machine-dependent and various random delays); 3) stalls due to lock contention; and 4) the resulting waiting time towards the barrier due to the critical threads.

We show cycle stacks of several epochs taken from various shared-memory programs. The examples aim at illustrating how thread criticality relates to each particular reason.

Case 1: Thread criticality due to different computational characteristics.

Figure 29 presents an individual epoch from the fmm application. The cycle stack shows that the time spent by the core during useful work varies significantly across the threads, while the varia-

[2]Note that heterogeneity can exist also in other levels of the system, for example in the cache and interconnect. In this work, we focus on single ISA core-level heterogeneity.

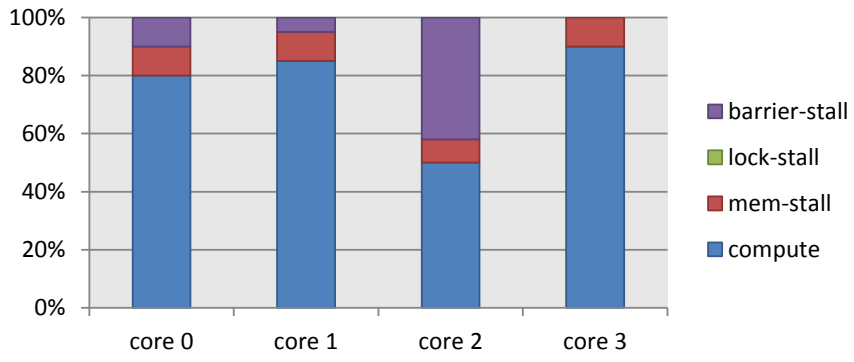

Figure 29: *Example of thread criticality caused by different computational characteristics.*

tions on memory access stalls and lock contention are less significant or do not exist. Consequently, the thread criticality in this particular case is very much affected by the compute-intensive time.

The variation in the cpu-time is basically caused by the different execution characteristics of each thread, presented to the cores (assuming identical cores). The different execution characteristics of each thread could emerge due to a different total instruction count (IC) that has to be executed by each core, the different type of instructions, the branch prediction accuracy, etc. Threads with vastly different characteristics are common in parallel programs that employ functional parallelism, such as pipelined programs and task parallelism. Nevertheless, significant cpu-time imbalances also occur often in SPMD (data parallel) programs, such as the one presented in this figure. Although the threads in SPMD programs are expected to execute similar code, poor data distribution to each thread can directly lead to such different computation demands. Also, although the threads execute the same part of the code, imbalances may occur from control statements that force each thread to follow a different execution path with different computational demands. As a result, threads may experience a different amount of computational load or different branch miss rates, resulting in a very different pipeline behavior.

Case 2: Thread criticality due to stalling on outstanding misses.

Processors often stall while waiting for outstanding misses to complete. Figure 30 presents an epoch from water-sp, in which memory stall cycles appear to occupy a significant amount of time that varies from thread to thread, while cpu-time and lock contention appear to have relatively low

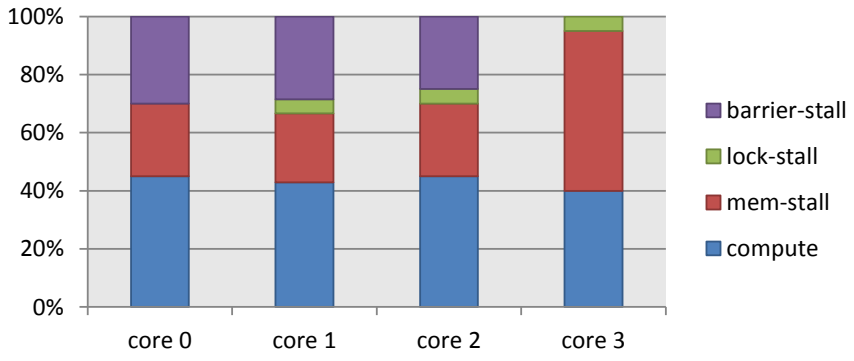

Figure 30: *Example of thread criticality caused by stalls on memory references.*

variability. The imbalance in this case is caused by the stalls created by the memory cycles.

As mentioned earlier, the variations in memory stall cycles could be due to application-specific characteristics, or random machine-dependent delays. For example, although all threads may be executing the same code, each thread may be operating on a data set with completely different locality characteristics (e.g., a cold loop vs a warm loop). Even when data and computation are exactly the same, machine artifacts such as the physical proximity of a core to its referencing data can cause variations on cache/memory miss latencies.

The amount and sensitivity of the memory stall cycles is also largely affected by how well the architecture hides long latencies. For example, an out-of-order core or a core with an aggressive prefetcher is expected to be less sensitive to the delays caused by misses. Therefore, imbalances in such systems are less likely to be related with the memory behavior. On the other hand, in-order cores experience longer memory stall times; thus, the cycle stack will be largely covered by the memory stall cycles, especially in memory bound applications.

Case 3: Thread criticality due to lock contention.

In the previous cases, stalls due to lock contention is marginal or not existent. This is common in most scientific applications where locks are sparsely used [114]. However, many other applications, such as databased and server applications, use locks heavily, which can lead to significant contention when entering critical sections, especially as the thread count increases.

Figure 31 shows the raytrace application, executed on 4 as well as on 8 parallel threads. Ray-

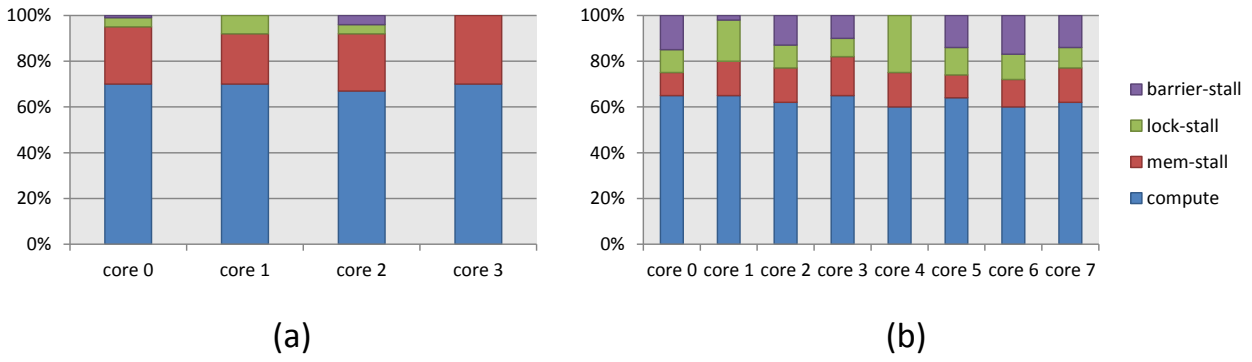

Figure 31: *Example of thread criticality caused by contention on locks.* Higher core count results to more lock contention, which affects the execution balance.

trace exhibits heavy use of locks, and lock contention increases as thread count grows. Although variations in cpu cycles and memory stall cycles exist, the imbalances are mainly caused by the time wasted in competing for locks, especially in the case of 8 threads. This example shows that capturing lock contention is crucial in correctly identifying the thread criticality in applications where locks are often in use.

Case 4: Thread criticality due to heterogeneous cores.

Figure 32a shows a well-balanced epoch taken from fft, running on a simulated system with 4 identical cores. The balance is achieved due to an effective static assignment of the load among the threads by the programmer, the homogeneity of the machine on which it runs, the minimum communication requirements, the non existence of locks, and the non existence of any other kind of interference from the system or the user level. Under these assumptions, the application runs ideally and as expected, without threads needing to wait for each other at the end.

Figure 32b shows the same application running on the same platform except that core 0 is running in double speed (frequency). The rapid execution of core 0 will execute the load in less time, and cause it to stall, waiting for the rest of the threads to finish. If the system is aware of the underlying heterogeneity, then such imbalances can be estimated, and dynamic scheduling could be used to close such gaps; however, depending on the task granularity, imbalances will still be visible.

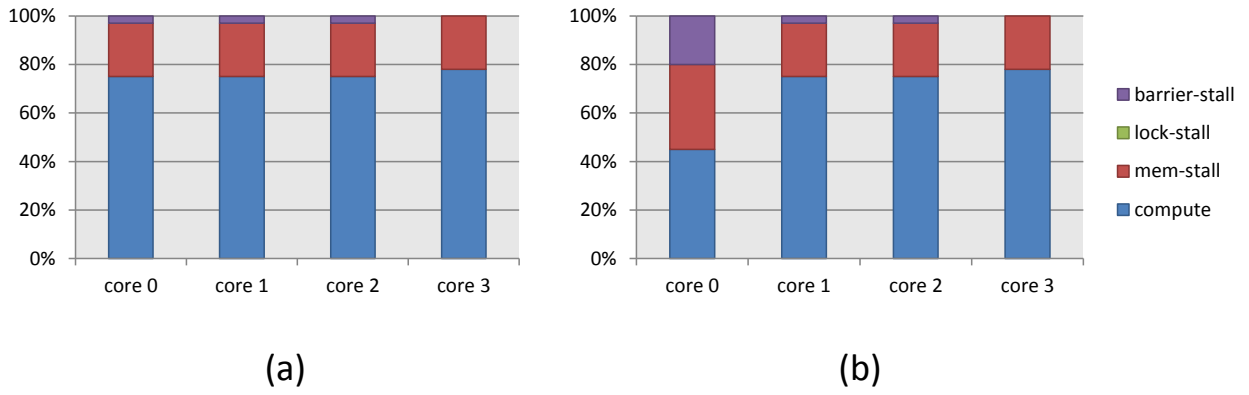

Figure 32: *Example of thread criticality caused by different core configurations.* Imbalance is created as core 0 runs on higher frequency.

4.4 PREDICTING THREAD CRITICALITY

4.4.1 Measuring Thread Criticality

The most critical thread always defines the critical path of the execution and is the one with the longest compute time; The compute time of a thread during an epoch is the time spent from the beginning of an epoch (release of a barrier), until the point where the thread hits the next barrier. The thread criticality order is simply the order of threads according to their compute time. Note that as the critical thread and the thread criticality order are naturally a property of an epoch, it is necessary to calculate thread criticality at each epoch.

For each thread t, the absolute compute time during epoch e can be expressed as:

$$T_{c(t,e)} = \text{IC}_{(t,e)} * \text{CPI}_{(t,e)} * \frac{1}{f_{(t,e)}} \tag{4.1}$$

where f the frequency of the core that the thread is running, IC the instruction count during the total execution of the epoch, and CPI the average clock per instruction during the epoch. Furthermore, the CPI can be decomposed to the sum of the individual portions of the CPI stack, following the cycle stack discussion in the previous section.

The time each thread t waits for the critical thread to finish is accordingly the difference between the compute time of the thread t and the compute time of the most cirtical thread. The ratio between these compute times is used to express the *relative criticality* of a thread t to the critical thread, or simply the *criticality* of the thread, i.e.,

$$\text{Criticality}_t = \frac{T_{c(t,e)}}{T_{c(\text{t-critical},e)}} \tag{4.2}$$

with the most critical thread being the one with max criticality (equal to 1).

If one or more factors in Equation 4.1 are insensitive across the threads, then the thread criticality is directly correlated with the remaining, sensitive factors. For example, if f, and IC are the same, then CPI will remain as the only factor related to the criticality. Then, decomposing the CPI to its stack components could reveal which particular metric(s) directly correlate with thread criticality.

4.4.2 Estimating Thread Criticality: State-of-the-Art

Predicting accurately the thread criticality before the end of the epoch is desirable as it opens the opportunity for further fine-grain dynamic load balancing and improvements in performance and energy. Typically, thread criticality predictors (TCPs) correlate criticality to some metric and predict the critical thread on the fly, or rely on history-based predictions that retrieve actual execution times from recorded previous execution instances. In what follows, we describe three different state-of-the-art TCPs and discuss some of their limitations. Then we introduce a new predictor (called SP-TCP) that combines barrier-based information with on-the-fly observations to achieve better overall predictability power. Table 11 lists the different thread criticality predictors discussed in this section.

IC-based TCP. In IC-based TCP, thread criticality is correlated to the number of instructions executed. Thus, for a given point in time, the thread which has executed the fewest instructions so far is the one that "lags behind", and hence the most critical. IC-based TCP works on the assumption that threads have equal amount of work to execute, i.e., the same number of instructions. In other words, IC-based TCP assumes that the factors IC_t and f_t in Equation 4.1 are the same for all t, and tracks the relative IC across threads to estimate the relative criticality.

REF. NAME	THREAD CRITICALITY METRIC
IC-based	relative instruction count
Miss-based	relative number of L1/L2 cache misses
Thrifty Barrier	absolute idle time based on barrier-history
SP-TCP.profile	relative compute time (based on relative CPI, f, and profiled barrier-based total IC)
SP-TCP	relative compute time (based on relative CPI, f, and runtime-predicted barrier-based total IC)

Table 11: List of Thread Criticality Predictors (TCPs) discussed and compared in this work.

Miss-based TCP. In miss-based TCP, thread criticality is correlated to the caches misses. Miss-based TCP works on the premise that a thread will have slow progress if has more misses to cope with. This is essentially similar to the IC-based TCP as it also tries to catch threads with lower CPI (more misses will cause more stall cycles). Hence, Miss-based TCP relies on the the same assumptions as IC-based, i.e., that the relative total IC across threads is the same. The Miss-based TCP model tracks both L1 and L2 misses, while giving more weight on L2 misses as those have higher latencies and will cause more idle cycles.

IC- and Miss- based TCPs are dynamic, metric-correlation-based predictors and work independent of epochs or any history information related to such granularity. In fact, the IC-based TCP was specifically proposed to work at the OS scheduling granularity, while the miss-based TCP on fixed time intervals. However, to make accurate and sensible observations and fairly compare them, we examine them at the epoch granularity.

Thrifty-barrier TCP. Thrifty barrier is different from the previous TCPs as it is based on history-based predictions. Thrifty barrier estimates the actual total execution time of barrier-intervals (epochs) based on the execution time of the previous invocation of the same intervals. Then, when a thread hits a barrier, the remaining idle time is calculated based on the total estimated time. Note, however, that although thrifty barrier predicts directly the waiting times for each thread, the predictions are performed after the first thread actually hits the barrier, and not ahead of time. Also note that the key element in thrifty barrier is the assumption that the different invocations of the epoch are expected to have equal absolute total execution time. Our work is closely related to

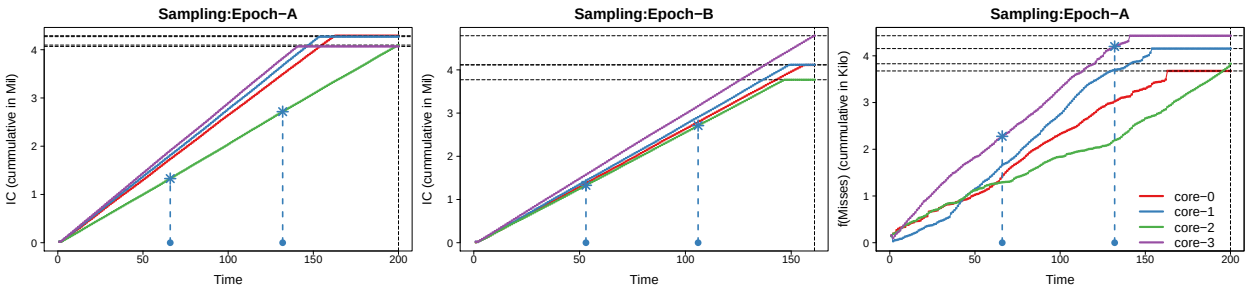

Figure 33: *Examples of how IC-based and Miss-based thread criticality metrics correlate with the actual critical thread during the execution of an epoch.* Plots (a) and (c) show an epoch in which the threads have a (relatively) balanced IC but Miss-based metric is not correlated. Plot (b) shows an epoch in which the total IC across threads is not balanced and thus IC-based metric is not correlated.

thrifty barrier as we also use the idea of barrier-based history information; however, we do not rely in predicting absolute execution time as such prediction is hard to work in general and is based on certain assumptions.

If thread criticality is correlated with a metric, then tracking the corresponding metric at run-time should be adequate for predicting the most critical thread. For example, Figure 33a shows that by measuring the instruction count of the executing threads and discovering the one with the minimum count (as suggested by IC-based TCP) we can correctly and consistently point to the critical thread. Thus, tracking the cumulative IC at any point during the epoch can be effectively used to directly predict the critical thread. Miss-based TCP is expected to work in a similar way; however the use of the Miss-based TCP for the same epoch (Figure 33c) leads instead to wrong predictions since the thread with the heaviest miss activity is not the slowest, for none of the points. The reason is probably the fact that in our example we use an out-of-order processor model. Out-of-order processors or other processors that try to hide the miss latency with various techniques reduce the effectiveness of the Miss-based TCP.

Figure 33b shows a different example where threads have unequal total IC to execute during the epoch. In such case, the criticality tends to move towards the IC factor of the Equation 4.1, and consequently the IC-based TCP fails to predict the critical thread. Generally speaking, any predictor which assumes that the total IC across threads is the same is more likely to fail in this case.

4.4.3 Estimating Thread Criticality: SP-TCP

In this work, we propose a new TCP, called Synchronization-Point Based Thread Criticality Predictor (SP-TCP) which collects both, history-based information from past epochs and on-the-fly measurements in the current epoch to build knowledge at the epoch granularity and predict the thread criticality order in the epoch before threads approach the next barrier. The goal of the SP-TCP is to predict all the factors in Equation 4.1 individually, and hence provide a robust and accurate estimation of criticality across different workloads and architectures.

The key observation behind the SP-TCP is that the IC factor in Equation 4.1 is predictable based on previous epochs, where the CPI factor can be estimated on-the-fly. To better explain this idea, we will walk through each of the factors in Equation 4.1 and discuss our methodology.

The frequency f in Equation 4.1 is the most predictable factor as it is given for free by the specifications of each operating core. When cores are allowed to change frequency dynamically and independently, it is expected that the frequency information is provided by hardware performance counters.

The IC is a factor that is basically application-dependent, and thus cannot be affected by any random delays or machine artifacts. Consequently, IC is likely to be an epoch property that has a predictable aspect when the epoch is replayed. In fact, observations indicate that threads either have the same amount of instructions to execute during an epoch, or, if they don't, their relative total IC is still predictable based on previous instances of the same epoch. Based on this observation, SP-TCP is designed to track the relative total IC across the threads during each epoch, store this information as a signature associated with each particular epoch, and use it to predict the relative IC between threads in future instances of the same epoch. This allows the SP-TCP to build knowledge about the relative total IC across the threads for the running epoch e at the very beginning of the epoch. The characterization details and results related to the IC predictability are given later in Section 4.5.

The remaining factor in Equation 4.1 is the CPI which is a machine-dependent factor and, as follows from the discussion in Section 4.3, it is hard to remain predictable across dynamic epoch instances. Thus, the CPI of the overall epoch is estimated on-the-fly based on early measurements of the epoch's CPI. Specifically, the CPI of the running epoch is monitored and is assumed to

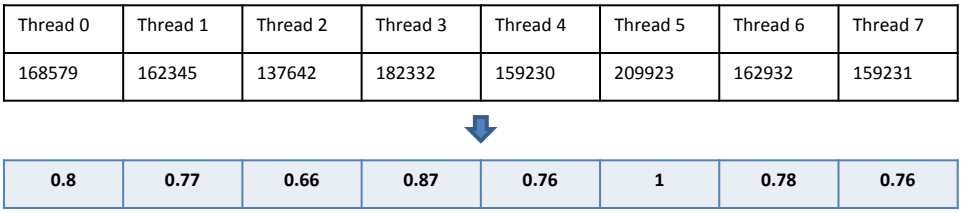

Figure 34: A signature holding the ICs of each thread is formed at the end of each epoch, and kept for hints in later epoch instances.

be representative after allowing some time from the beginning of the epoch. The CPI predictor can build confidence on its decision by re-evaluating the CPI periodically while the epoch is still running, till it finishes.

Storing history information and performing predictions. SP-TCP requires operations and mechanisms that can store and retrieve epoch-wide information, periodically monitor certain metrics, and perform prediction actions. We assume that on each core, the f, IC, and CPI are available to the predictor at any given point in time. These metrics are typically tracked and kept in hardware performance counters and thus can be readily available. The counters are reset at the beginning of each epoch in order to reflect statistics always at the epoch granularity.

When a thread hits a barrier point, the accumulated IC up to that point is retrieved and temporarily stored into a dedicated counter. When all the threads have reached the barrier (i.e., at the end of the epoch), the ICs are collected to form an IC-signature of the epoch. This signature is basically a vector of counters (as many as the threads) indicating the IC for each thread during the epoch. The individual ICs are then normalized to the maximum IC to form the *relative* IC-signature. Note that the thread with the maximum IC is not necessarily the critical thread. Figure 34 illustrates how an IC-signature is conceptualized. The signatures are stored into an SP-table, similar to the one described in Section 2.5.

When a new dynamic instance of the same epoch is encountered, the stored signatures of the corresponding epoch are retrieved from the SP-table. Given that the frequency f is also available, the remaining factor to estimate is the CPI. The epoch's accurate CPI is is not known before the end of the epoch; however, as the epoch progress, we can measure the CPI up to the current execution

and use it as an estimation for the whole epoch. We allow 250K cycles[3] before evaluating the CPI of each thread for the first time within each epoch. The CPI of each thread is collected into a vector to form the relative CPI-signature, similar in concept to the IC-signature. Having now all factors in Equation 4.1 available, we are able to calculate the criticality of each thread, and form the criticality order or/and find the thread that is expected to be the most critical. The predictor can evaluate whether the criticality is significant, e.g., whether the most critical thread is significantly slower than the rest, and enable the appropriate optimization.

With the expectation that each thread will exhibit the same average CPI during the epoch, a prediction action performed as early as 250K cycles after the beginning of the epoch would provide an accurate estimate. To further build confidence towards the correctness of the initial prediction, we re-evaluate the CPIs periodically while the epoch progress to ensure that the average CPIs remain stable. In the case where no history information is available to the executing epoch (i.e., the first instance of the epoch is executing), the criticality predictor assumes that the relative IC across the threads during the epochs will be the same. Consequently, the SP-based thread criticality predictor in such cases becomes similar to the IC-based TCP, and predictions are made based on the the relative CPI and f.

4.5 EVALUATION

4.5.1 Methodology

We evaluate SP-TCP in comparison to the IC-based and Miss-based TCPs. First, we evaluate each approach based on how effective is in predicting the most critical thread given that a prediction action can be perform at any point during the execution of the epoch. Then, we separately evaluate the effectiveness of epoch's IC predictor, which is a distinct component of the SP-TCP. Lastly, we demonstrate the overall effectiveness of SP-TCP by using it to dynamically identify and accelerate critical threads and improve overall performance.

We present results for both, profile-based and runtime-based SP-TCP. In profile-based SP-TCP,

[3] If cores have different frequency, we assume the fastest core defines the "global" clock.

the relative IC-signature for each epoch is known a priori; in runtime-based SP-TCP, the relative IC-signature is predicted at runtime based on the epoch's history.

To accelerate critical threads for performance, a TCP dynamically detects the most critical thread in each epoch before the barrier is reached, and then boosts the operating frequency of the core on which the thread is running. Although it is possible to accelerate more than one critical threads, we restrict our evaluation on scaling the frequency only for the most critical thread. Note that accelerating a thread is also possible through other techniques such as migrating it to a faster core. We pick frequency scaling because it is a simple and adequate technique to demonstrate the effectiveness of SP-TCP.

The thread criticality prediction action is performed at 250K cycles after the beginning of each epoch. If a signature from a previous epoch is available, then the prediction will take into consideration the retrieved relative IC-signature. The outcome of the prediction will be an estimation of the relative criticality of each thread during the epoch. The frequency f of each thread is then tuned accordingly in order to eventually balance the compute times.

The dynamic frequency scaling mechanism relies on the ability to frequently change the voltage and frequency of individual cores. On-chip voltage regulators may take hundreds to thousands of cycles to change voltage. If frequency changes infrequently (e.g., at most once during each epoch), then such delays are not expected to be significant, as epochs are relatively coarse-grain intervals. Allowing more frequent frequency/voltage scaling (e.g., more than once during the epoch) might require faster regulators. Previous works have shown that transitions times as fast as 7ns are actually possible. In our simulations, we assume a fixed 1,000 cycles for each transition.

Because the frequency of a core scales only in discrete levels, we must decide the scale of the adaptive action based on the relative criticality of the most critical thread to the next critical. Specifically, assuming that frequency levels can change by 0.2 GHz on a 2GHz core, we take a different action whenever the predicted relative compute times differ in multiples of 0.1. Therefore, if for example, the compute time of the most critical thread to the one of the next critical is, let's say 1.1, then the most appropriate actions would be to boost the critical core to the next frequency level (i.e., 2.2GHz).

To perfectly balance the relative criticality, we must (apart from predicting correctly the relative compute time) execute the whole computation part in the correct new calculated frequency. In other

Model	*Configuration Values*
Cores	4 out-of-order cores, dispatch width 4, window size 128, 10 outstanding load/stores .
L1 I Cache	32 KB, 4-way, 1-cycle load-to-use latency
L1 D Cache	32 KB, 8-way, 1-cycle load-to-use latency
L2 Cache	256 KB, 8-way , 3-cycle tag-access latency
L3 Cache	8MB shared, 16-way, 10-cycle tag-access latency
Coherence	MESI protocol, Directory
NoC	Crossbar
Main mem.	150-cycle latency, 4KB pages, 64-entry 4-way TLBs

Table 12: Simulated machine architecture configuration.

words, it is essential to perform the frequency adaptation at the beginning of the epoch–a situation that is in conflict with the need to spend time tracking CPI of the epoch. In our evaluation, we perform the first prediction and frequency scaling 250K cycles after the beginning of each epoch. Such interval is usually safe as it is short (relative to the whole execution of the epoch), and adequate to collect a good approximation for the CPI. We re-evaluate the prediction periodically while the epoch runs to build confidence towards the prediction (every 500K instructions). Re-evaluation involves retrieving and comparing once again the CPI. If the relative CPI appears to change significantly, then the frequency is adjusted accordingly.

4.5.2 Methodology

4.5.3 Simulation Environment

We evaluate the proposed scheme using the sniper simulation infrastructure [18]. We simulate 4-core and 8-core tile based CMP. Each core resembles an out-of-order execution pipeline that is modeled after an interval-based modeling methodology [18]. The cache organization and parameters are listed in Table 12. The benchmarks are taken from the Splash2 and PARSEC suites [119, 13]. Details for the benchmarks can be found in Section 2.3.

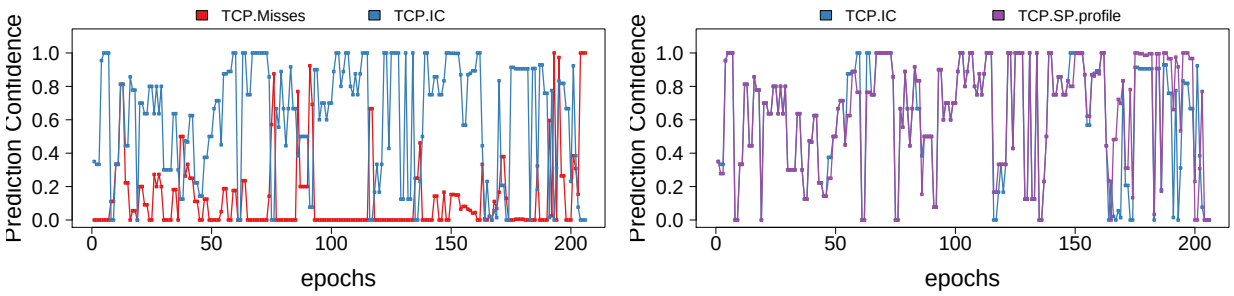

Figure 35: *Prediction accuracy for different prediction approaches*. Accuracy measures the probability to predict correctly the critical thread while the epoch is running.

4.5.4 Prediction Accuracy

We compare the profile-based SP-TCP with the IC-based and Miss-based predictors in terms of how consistent they are in predicting accurately the critical thread. Specifically, we measure the probability to predict correctly the critical thread at any point during the execution of an epoch. To measure this quantity, we apply a prediction sampling methodology that follows from the examples given in Figure 33. If all the samples correctly identify the critical thread, then the correlation is perfect and the prediction accuracy is 1.0. Figure 35 shows the results for each different epoch, for a large number of epochs taken from various programs. Figure 36 summarizes the results by showing the average measurements across the epochs for each individual program.

The results in Figure 35 (left) indicate that for most of the epochs, the IC-based predictor is much more accurate than the Miss-based predictor. Figure 35 (right) shows that the SP-TCP improves the prediction accuracy for those epochs in which the relative IC across threads is not equal (epochs towards the left end of the figure). In contrast, for epochs that have balanced-IC (epochs towards the right side of the figure), the SP-TCP is basically equivalent to the IC-based TCP and hence does not provide any additional value. In fact, the results in this particular case reflect the actual effectiveness of using CPI sampling to predict the relative CPI between the threads for the overall epoch. As one can observe, although the prediction has good accuracy for most epochs, there are cases where the CPI sampling cannot capture the actual critical thread (i.e., CPI sampling is not representative and thus not good for prediction). There are two different reasons

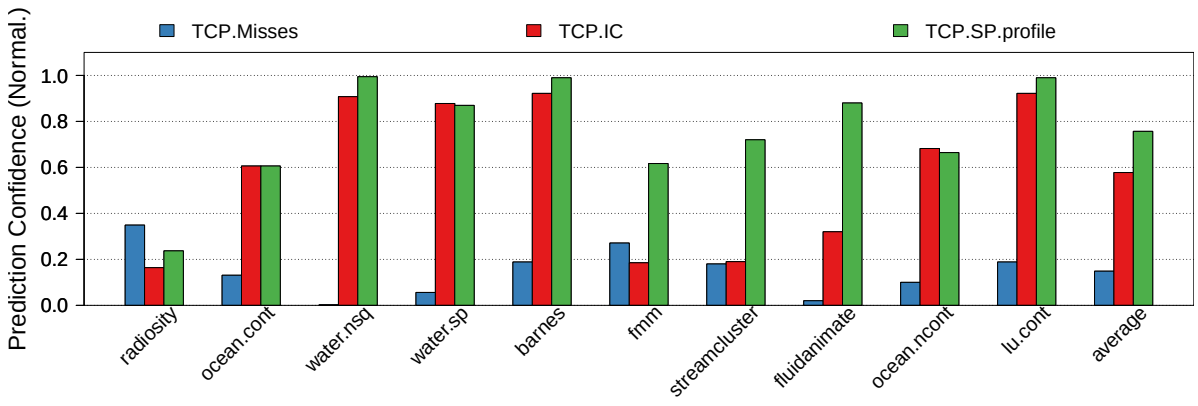

Figure 36: *Prediction accuracy (average results per benchmark).* Accuracy measures the probability to predict correctly the critical thread while the epoch is running.

that can cause this inaccuracy. The first one, which is the most common, is due to the fact that oftentimes the thread criticality is insignificant (i.e., all threads reach the barrier almost at the same time), In this case it is very hard for the predictor to catch the thread that will actually hit the barrier last, as the relative difference on the sampled CPI metric is not significant enough. The second reason could be simply the fact that sampling the CPI during the epoch is not a representative measure of the CPI that characterizes the overall epoch. This can happen if the threads exhibit very different execution behavior towards the very end of the epoch.

Figure 36 shows that, on average, predicting the critical thread at run-time using the approach suggested by the SP-TCP increases the probability for a correct prediction. The Miss-based prediction is on average poor compared to the IC-based. Although this contradicts with the conclusions made in [12], we believe is due to the fact that in this study we use an out-of-order processor model. As mentioned also by the authors in [12], the Miss-based predictor is better suited for simple in-order cores.

4.5.5 Predicting Epoch's Total IC

As we discussed so far, the SP-TCP predictor must be able to estimate the total IC of each epoch. While a profile-run can be proved adequate in providing this information in static environments, a fully dynamic SP-TCP must be able to predict this information at run-time. In this section, we

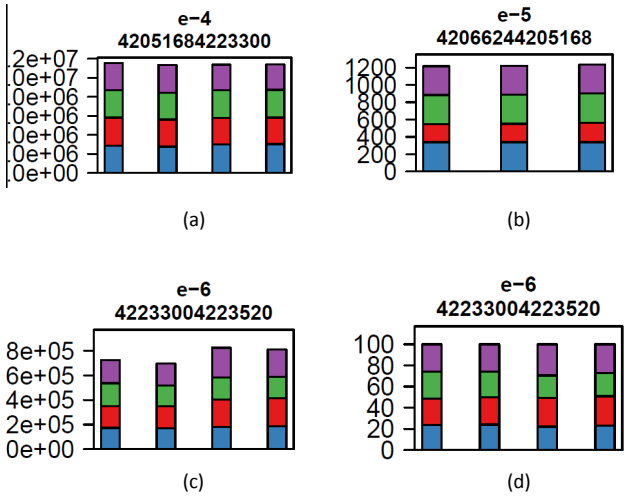

Figure 37: *Number of instructions executed in each thread at the epoch granularity.* Different colors represent different threads and different bars represent different instances of the same epoch. (a) balanced IC; (b) non balanced IC but the absolute ICs are stable across epoch instances; (c) non balanced IC but the relative ICs (as shown in (d)) are stable across epoch instances;

evaluate a simple approach that exploits the repeatability of barriers to predict the IC information necessary for the predictions.

We characterize the IC across the threads within each epoch, as well as across the dynamic instances of each epoch. We categorize our observations to four cases:

Balanced-IC across the threads during an epoch. Oftentimes, all the threads execute the same amount of instructions during an epoch. Figure 37a shows such examples. Each bar in this figure represents an epoch instance, and the breakdown the IC executed by each of the 4 threads. The equal split indicates a balanced IC across the threads.

Imbalanced-IC across the threads during an epoch. In contrast to balanced-IC, there are other cases where the IC executed by each thread during an epoch is different. For example, as shown in Figure 37b, the second core (in red) executes less instructions than the other threads during the epoch. Overall, the IC across the threads is a property that is solely defined by the software. Although in data parallelism it is common for software to optimize for the same IC across threads, we cannot generalize that threads are expected to have the same IC during an epoch, and hence

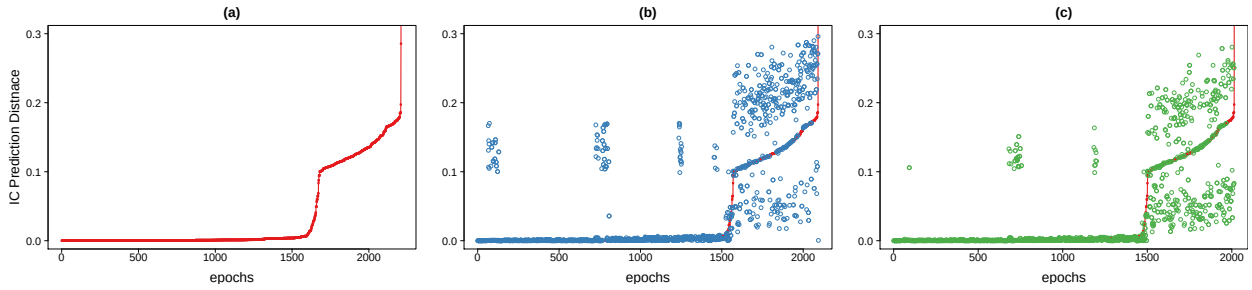

Figure 38: *Thread relative IC prediction effectiveness at the epoch granularity*. (a) equal IC prediction, (b) last epoch prediction, (c), last epoch + stride2 prediction. The closer the distance to zero, the more similar the predicted IC vector is to the actual one.

such imbalances are possible.

Stable relative-IC patterns across the dynamic instances of an epoch. We investigate whether the IC between the threads can be predicted based on history from previous invocations of the same epoch. We observe two stable relative-IC patterns that are common across the dynamic instances of epochs.

Last-epoch relative-IC: The relative-IC in a given epoch is the same as the one in the previous instance of that epoch. This continuously stable relative-IC is observed for both balanced-IC and imbalance-IC epochs. A simple last-value history-based predictor can capture such stable behavior.

Stride-2 pattern for relative-IC: In this case, the same relative-IC reoccurs every other dynamic instance of the epoch. Epochs with many instances may even have repeatable patterns in a higher distance than 2-stride. Designing a predictor to capture any stride-X pattern is straightforward, however the storage requirements of such predictor increases linearly with X. Thus, we restrict our predictor to stride-2.

Unpredictable: Although it is possible for other patterns to exist, we consider every other case unpredictable.

Figures 38 and 39 compare how effective are the above predictors in predicting the epochs' relative-IC, for each epoch and for each benchmark on average respectively. The relative-IC is represented by a vector, and the prediction effectiveness is measured by how similar/different is

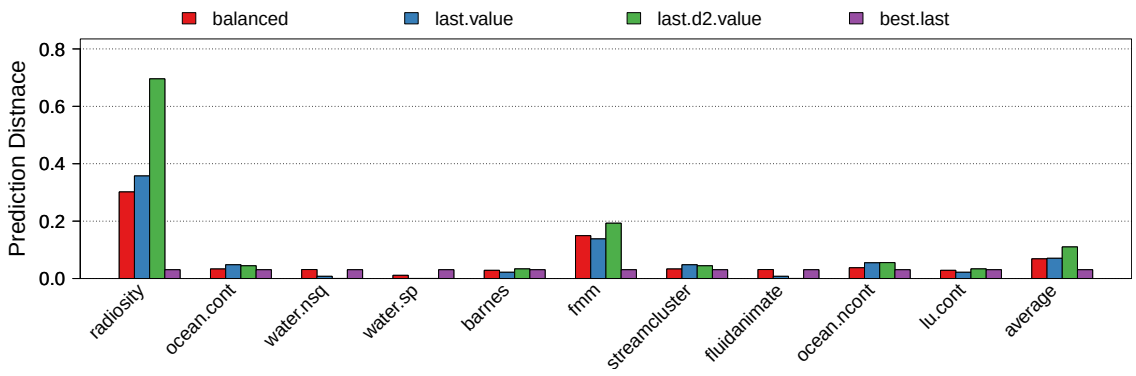

Figure 39: *Relative IC prediction effectiveness (average results per benchmark).*

the predicted vector from the actual vector.

Figure 38a illustrates that many epochs have simply IC-balanced. For these cases, prediction is not really necessary and this is the main reason most TCPs operate under this assumption. The last-epoch relative-IC prediction can help in predicting accurately some of the epochs that are not IC-balanced, as shown in Figure 38b. However, there are epochs that do not follow a last-epoch stable pattern, and therefore forcing a last-epoch prediction could affect reversibly the prediction effectiveness. Figure 38c shows that, if the stride-2 pattern is included in the predictions, accuracy is improved for many of the epochs. The epochs for which the prediction is still ineffective are epochs which may follow different IC behavior not explored by the predictors presented here, or are inherently unpredictable.

The average results in Figure 39 suggest that the history-based predictions can improve the predictability of the epochs' relative-IC in each application.

4.5.6 Performance Results

Figure 40 shows the overall performance improvements of the SP-TCP, assuming a DVFS is used to accelerate the core running the predicted most critical thread. The level of acceleration is always performed according to the speed of the next predicted least critical thread (i.e., the frequency level is chosen to match the speed of the next critical). Note that if the prediction is incorrect, it is impossible to gain in performance as the critical path will remain unaffected.

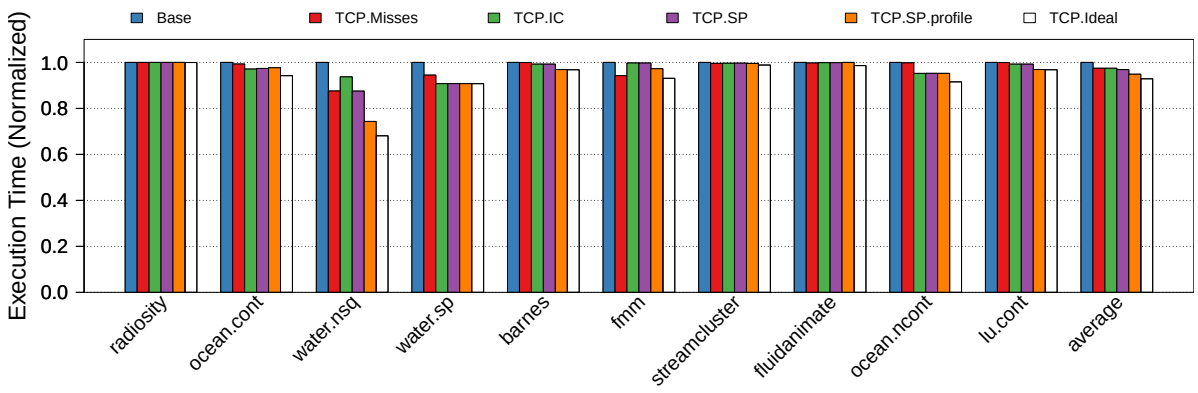

Figure 40: *Performance improvement as a result of accelerating the predicted critical thread.* Predictions are perform at the epoch granularity. Improvements are possible only if the critical thread is predicted correctly.

Results show 8% and 10% execution time improvement on average for SP-TCP predictors with dynamic and profile-based IC-relative prediction respectively. In water.nsq the large gap between the critical thread and the next one is eliminated gaining significant in performance. Benchmarks like radiosity and fft do not show any gains as these benchmarks appear to be well balanced in all perspectives. Lu gains little relative to its total potential gain because in some epochs the relative IC is not stable across the instances, hence the critical thread is often wrongly predicted. We found that the relative IC in lu follows a stride pattern, and thus can be correctly predicted if we consider more aggressive predictors, as done in Chapter 3. In certain applications, the improvements appear to follow closer the ideal case–which is achievable when the speed of the critical thread is perfectly adjusted to the next slower thread in the criticality order (rightmost bar).

When achieving better execution balance, we gain not only in execution speed, but also in the amount of stall cycles spent/wasted on the parallel processing resources. Figure 41 shows the reduction of total wasted cycles due to accurate prediction and acceleration of the critical thread. The reductions in stall cycles roughly translate to energy savings during stall times. However, they do not indicate the actual energy savings as frequency scaling affects the way energy is consumed during execution.

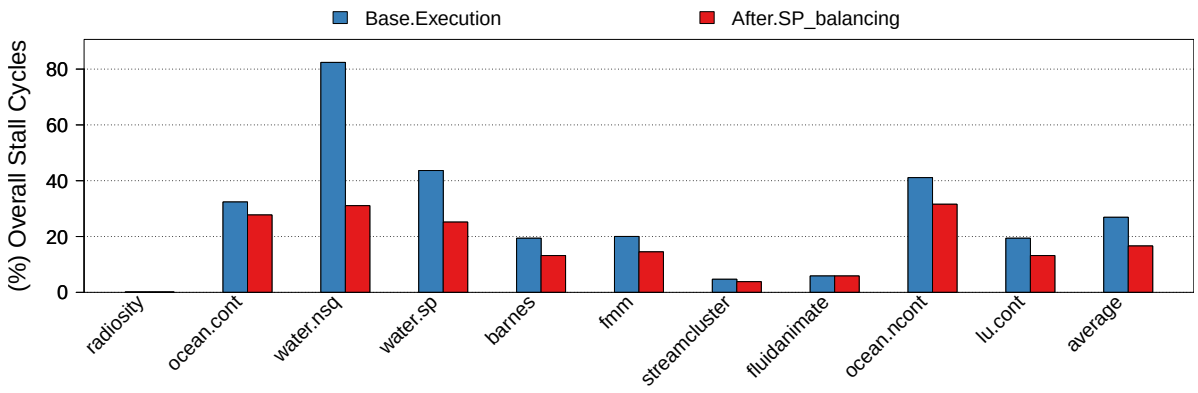

Figure 41: *Total reduction in wasted cycles at barriers.* The metric measures the percentage over the total amount of cycles needed to complete the execution of parallel section.

4.6 SUMMARY

Synchronization points often cause threads to wait for each other, stalling program progress and wasting system resources. Understanding the relative progress of threads and predicting the critical ones at run-time has important ramifications for performance and resource management. This chapter has examined the execution behaviors that lead to thread criticality, and has identified the basic limitations of current thread criticality predictors using existing multithreaded applications. To improve the predictability of critical threads, this work builds on previous techniques and proposes a more robust and general predictor. By accurately accelerating the most critical thread, we are able to improve performance and reduce the amount of time threads waste cycles waiting.

5.0 CONCLUSIONS

As transistor density continues to grow, it is apparent that the scaling of the number of on-chip cores would follow proportionally. The abundance of the thread level parallelism and the promotion of the shared memory model by the CMPs has led the software industry to establish the shared memory paradigm not only as the most standard programming practice, but also as the only way of gaining in performance. Thread synchronization and communication will remain fundamental to this practice, independent of whether their abstraction or implementation level will shift in different directions.

This thesis examines the time-varying execution characteristics of shared-memory parallel applications in conjunction to the synchronization points that exist in these applications. Through workload-driven evaluation and extensive simulation studies, we demonstrate that synchronization points can be exploited to effectively guide on-chip resource management techniques, from dynamically scaling the frequency/voltage of the network-on-chip, to dynamically predicting communication patterns, and to dynamically detecting and eliminating critical execution paths.

As synchronization points are application-injected events, they allow us to form an optimization and resource management framework that is simple and flexible in implementation, is independent of the architecture, has low space overheads, and is in a sense aware of the communication and race conditions between the threads. Consequently, it enables a simple and effective application-driven approach for tracking and exploiting various execution characteristics of a parallel program at run time.

We anticipate that this research provides a novel and valuable framework that can be used as a basis for further performance optimization and resource management in CMPs, as well as a guide in understanding important time-varying execution characteristics in multithreaded applications.

BIBLIOGRAPHY

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture*, HPCA, 1997.

[2] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner prediction for accelerating cache-to-cache transfer misses in a CC-NUMA architecture. In *Proc. of Conf. on Supercomputing*, SC, 2002.

[3] M. E. Acacio, J. González, J. M. García, and J. Duato. The use of prediction for accelerating upgrade misses in CC-NUMA multiprocessors. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT, 2002.

[4] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, 2013.

[5] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 2003.

[6] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, 2005.

[7] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 2009.

[8] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO'00, 2000.

[9] A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore, and G. J. M. Smit. An energy and performance exploration of network-on-chip architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 2009.

[10] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of SPLASH-2 and PARSEC. In *Proc. Int'l Symp. on Workload Characterization*, IISWC, 2009.

[11] N. Barrow-Williams, C. Fensch, and S. Moore. Proximity coherence for chip multiprocessors. In *Proc. Int'l Conf. on Parallel Aarchitectures and Compilation Techniques*, PACT, 2010.

[12] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, 2009.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Princeton University Technical Report TR-811-08*, 2008.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT, 2008.

[15] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast snooping: a new coherence method using a multicast address network. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 1999.

[16] J. A. Brown, R. Kumar, and D. Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *Proc. Int'l Symp. on Parallel Algorithms and Architectures*, SPAA, 2007.

[17] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, 2008.

[18] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[19] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proc. Int'l Symp. on Operating Systems Principles*, SOSP, 1991.

[20] L. Cheng, J. B. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *Proc. of the Int'l Symp. on High Performance Computer Architecture*, HPCA '07, 2007.

[21] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proc. Int'l Symp. on Microarchitecture*, MICRO, 2006.

[22] L. Choi and P. C. Yew. A compiler-directed cache coherence scheme with improved intertask locality. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Supercomputing '94, 1994.

[23] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, ISCA '93, 1993.

[24] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 1997.

[25] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, 2011.

[26] S. Demetriades and S. Cho. Barrierwatch: characterizing multithreaded workloads across and within program-defined epochs. In *Proc. of the 8th ACM Int'l Conf. on Computing Frontiers*, CF '11, 2011.

[27] S. Demetriades and S. Cho. Predicting coherence communication by tracking synchronization points at run time. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, 2012.

[28] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. *SIGARCH Computer Architecture News*.

[29] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[30] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, 2003.

[31] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *Proc. of the 44th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, MICRO-44 '11, 2011.

[32] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *Proc. Int'l Symp. on Microarchitecture*, MICRO, 2006.

[33] M. Ekman, P. Stenström, and F. Dahlgren. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proc. of the 2002 Int'l Symp. on Low power electronics and design*, ISLPED '02, 2002.

[34] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Proc. Int'l Symp. on Microarchitecture*, MICRO, 2008.

[35] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. of the 38th annual Int'l Symp. on Computer architecture*, ISCA '11, 2011.

[36] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, 2005.

[37] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, 1998.

[38] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, 2005.

[39] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. Erasing core boundaries for robust and configurable performance. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.

[40] S. Herbert and D. Marculescu. Variation-aware dynamic voltage/frequency scaling. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009.

[41] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: software and hardware for scalable multiprocessors. *ACM Trans. Comput. Syst.*, November 1993.

[42] M. J. Hind, V. T. Rajan, M. Hind, V. T. Rajan, P. F. Sweeney, and P. F. Sweeney. Phase shift detection: A problem classification. In *IBM Research Report #22887*, 2003.

[43] H. Hossain, S. Dwarkadas, and M. C. Huang. Improving support for locality and fine-grain sharing in chip multiprocessors. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT, 2008.

[44] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010.

[45] D. C. Howell. *Fundamental Statistics for the Behavioral Sciences*. Duxbury Resource Center, 4th edition, 1998.

[46] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, 2003.

[47] http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/. Java concurrency utilities.

[48] http://quid.hpl.hp.com:9081/cacti/. CACTI 5.3.

[49] http://www.amd.com. Amd inc.

[50] http://www.intel.com. Intel co.

[51] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi detail.html. Intel xeon phi product family.

[52] http://www.tilera.com/products/processors/TILE-Gx-Family. Tilera TILE-Gx processor family.

[53] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. *SIGARCH Comput. Archit. News*, 2003.

[54] IEEE Computer Society. IEEE standard for scalable coherent interface (SCI). 1992.

[55] Intel Co. MESIF protocol. US Patent 6922756.

[56] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra. Phase-based application-driven hierarchical power management on the single-chip cloud computer. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT, 2011.

[57] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 2003.

[58] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, 2002.

[59] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti. Circuit-switched coherence. In *IEEE 2nd Network on Chip Symp.*, NOCS, 2008.

[60] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, 2012.

[61] S. Kaxiras and J. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proc. Int'l Symp. on High-Performance Computer Architecture*, HPCA, 1999.

[62] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proc. Int'l Symp. on High-Performance Computer Architecture*, HPCA, 2000.

[63] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, 2012.

[64] E. J. Kim, G. M. Link, K. H. Yum, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and C. R. Das. A holistic approach to designing energy-efficient cluster interconnects. *IEEE Transactions on Computers*, 2005.

[65] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, 2003.

[66] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 2003.

[67] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *Proc. Int'l Conf. on Supercomputing*, ICS, 1995.

[68] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, 2007.

[69] A. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 1999.

[70] A. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 2000.

[71] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[72] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, 2006.

[73] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, 2004.

[74] J. Laudon and D. Lenoski. The SGI Origin: A CC-NUMA highly scalable server. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 1997.

[75] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 1995.

[76] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 1984.

[77] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2009.

[78] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, 2009.

[79] S. Leventhal and M. Franklin. Perceptron based consumer prediction in shared-memory multiprocessors. In *Int'l Conf. on Computer Design*, ICCD, 2006.

[80] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, 2004.

[81] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, 2010.

[82] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, 2006.

[83] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting barriers to optimize power consumption of cmps. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005.

[84] W. Liu and M. C. Huang. Expert: expedited simulation exploiting program behavior repetition. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, 2004.

[85] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003.

[86] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 2002.

[87] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling, 2000.

[88] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 2003.

[89] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence. decoupling performance and correctness. In *Proc. of the 30th Annual Int'l Symp. on Computer Architecture*, ISCA, 2003.

[90] A. K. Mishra, R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C. R. Das. A case for dynamic frequency tuning in on-chip networks. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[91] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 2005.

[92] S. Mukherjee and M. Hill. Using prediction to accelerate coherence protocols. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 1998.

[93] J. Nilsson, A. Landin, and P. Stenstrom. The coherence predictor cache: a resource-efficient and accurate coherence prediction infrastructure. In *Proc. of the Int'l Parallel and Distributed Processing Symp.*, IPDPS, 2003.

[94] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, 2007.

[95] E. Perelman, M. Polito, J. yves Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *In International Parallel and Distributed Processing Symposium*, IPDPS'06, 2006.

[96] P. Ratanaworabhan and M. Burtscher. Program phase detection based on critical basic block transitions. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, 2008.

[97] J. Reinders. Intel threading building blocks, 2007.

[98] A. Ros, M. E. Acacio, and J. M. Garcia. A direct coherence protocol for many-core chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 2010.

[99] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, 2007.

[100] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, 2009.

[101] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, H. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *In Proc. of the 35th Micro*, 2002.

[102] L. Shang, L.-S. Peh, and N. K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, 2003.

[103] S. Shao, A. K. Jones, and R. Melhem. Compiler techniques for efficient communications in circuit switched networks for multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2009.

[104] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS XI, 2004.

[105] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGARCH Comput. Archit. News*, 2002.

[106] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. *SIGARCH Comput. Archit. News*, 2003.

[107] Y. Solihin. *Fundamentals of Parallel Computer Architecture*. Solihin Books, 1st edition, 2009.

[108] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *Workshop on Memory Performance Issues*, WMPI, 2004.

[109] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Transactons on Parallel and Distributed Systems*, 2002.

[110] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, 2006.

[111] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. of the Int'l Symp. on Computer Architecture*, ISCA '93, 1993.

[112] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Op. Syst.*, ASPLOS, 2009.

[113] K. T. Sundararajan, T. M. Jones, and N. Topham. A reconfigurable cache architecture for energy efficiency. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, 2011.

[114] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, 2010.

[115] Tilera Co. and http://www.tilera.com. Tilera TILE64 processor.

[116] P. Trancoso and J. Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *Proc. Int'l Conf. on Parallel Processing*, ICPP, 1996.

[117] F. Vandeputte and L. Eeckhout. Phase complexity surfaces: characterizing time-varying program behavior. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, 2008.

[118] F. Vandeputte, L. Eeckhout, and K. De Bosschere. A detailed study on phase predictors. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, 2005.

[119] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. Int'l Symp. on Computer Architecture*, ISCA, 1995.

[120] Y. Zhang, B. Ozisikyilmaz, G. Memik, J. Kim, and A. Choudhary. Analyzing the impact of on-chip network traffic on program phases for cmps. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009.